Apple

Apple Numerics Manual

Part I: The Standard Apple Numeric Environment

Part II: The 6502 Assembly-Language SANE Engine

Part III: The 68000 Assembly-Language SANE Engine

WORK-BENCH

030-0247-A

Table of Contents

List of Tables

Chapter 2 Data Types

Chapter 4 Conversions

Chapter 7 Infinities, NaNs, and Denormalized Numbers

Appendix C 6502 SANE Quick Reference Guide

Appendix C 68000 SANE Quick Reference Guide

Part I: The Standard Apple Numeric Environment

Table of Contents

Chapter 1

Introduction

This manual describes the Standard Apple Numeric Environment (SANE). Apple supports SANE on several current products and plans to support SANE on future products. SANE gives you access to numeric facilities unavailable on almost any computer of the early 1980's—from microcomputers to extremely fast, extremely expensive supercomputers. The core features of SANE are not exclusive to Apple; rather they are taken from Draft 10.0 of Standard 754 for Binary Floating-Point Arithmetic [10] as proposed to the Institute of Electrical and Electronics Engineers (IEEE). Thus SANE is one of the first widely available products with the arithmetic capabilities destined to be found on the computers of the mid-1980's and beyond.

The IEEE Standard specifies standardized data types, arithmetic, and conversions, along with tools for handling limitations and exceptions, that are sufficient for numeric applications. SANE supports all requirements of the IEEE Standard. SANE goes beyond the specifications of the Standard by including a data type designed for accounting applications and by including several high-quality library functions for financial and scientific calculations.

IEEE arithmetic was specifically designed to provide advanced features for numerical analysts without imposing an extra burden on casual users. (This is an admirable but rarely attainable goal: text editors and word processors, for example, typically suffer increased complexity with added features, meaning more hurdles for the novice to clear before completing even the simplest tasks.) The independence of elementary and advanced features of the IEEE arithmetic was carried over to SANE.

Throughout this manual, references in brackets are to the annotated bibliography in Part I. Words printed in bold type are defined in the glossary in Part I.

Chapter 2

Data Types

SANE provides three **application** data types (single, double, and comp) and the **arithmetic** type (extended). Single, double, and extended store floating-point values and comp stores integral values.

The **extended** type is called the arithmetic type because, to make expression evaluation simpler and more accurate, SANE performs all arithmetic operations in extended precision and delivers arithmetic results to the extended type. **Single**, **double**, and **comp** can be thought of as space-saving storage types for the extended-precision arithmetic. (In this manual, we shall use the term *extended precision* to denote both the extended precision and the extended range of the extended type.)

All values representable in single, double, and comp (as well as 16-bit and 32-bit integers) can be represented exactly in extended. Thus values can be moved from any of these types to the extended type and back without any loss of information.

Choosing a Data Type

Typically, picking a data type requires that you determine the trade-offs between

- fixed- or floating-point form
- precision
- range
- memory usage
- speed.

The precision, range, and memory usage for each SANE data type are shown in Table 2-1. Effects of the data types on performance (speed) vary among the implementations of SANE. (See Chapter 4 for information on conversion problems relating to precision.)

Most accounting applications require a counting type that counts things (pennies, dollars, widgets) exactly. Accounting applications can be implemented by representing money values as integral numbers of cents or mils, which can be stored exactly in the storage format of the **comp** (for computational) type. The sum, difference, or product of any two comp values is exact if the magnitude of the result does not exceed $2^{63} - 1$ (that is, 9,223,372,036,854,775,807). This number is larger than the U.S. national debt expressed in Argentine pesos. In addition, comp values (such as the results of accounting computations) can be mixed with extended values in floating-point computations (such as compound interest).

Arithmetic with comp-type variables, like all SANE arithmetic, is done internally using extended-precision arithmetic. There is no loss of precision, as conversion from comp to extended is always exact. Space can be saved by storing numbers in the comp type, which is 20 percent shorter than extended. Non-accounting applications will normally be better served by the floating-point data formats.

Values Represented

The floating-point storage formats (single, double, and extended) provide binary encodings of a **sign** (+ or -), an **exponent**, and a **significand**. A represented number has the value

$\pm \text{significand} * 2^{\text{exponent}}$

where the significand has a single bit to the left of the binary point (that is, $0 \leq \text{significand} < 2$).

Range and Precision of SANE Types

This table describes the range and precision of the numeric data types supported by SANE. Decimal ranges are expressed as chopped two-digit decimal representations of the exact binary values.

Table 2-1 *SANE Types*

Type class	**Application**			**Arithmetic**
Type identifier	Single	Double	Comp	Extended
Size (bytes:bits)	4:32	8:64	8:64	10:80
Binary exponent range				
Minimum	-126	-1022	--	-16383
Maximum	127	1023	--	16383
Significand precision				
Bits	24	53	63	64
Decimal digits	7-8	15-16	18-19	19-20
Decimal range (approximate)				
Min negative	-3.4E+38	-1.7E+308	≅-9.2E18	-1.1E+4932
Max neg norm	-1.2E-38	-2.3E-308		-1.7E-4932
Max neg denorm†	-1.5E-45	-5.0E-324		-1.9E-4951
Min pos denorm†	1.5E-45	5.0E-324		1.9E-4951
Min pos norm	1.2E-38	2.3E-308		1.7E-4932
Max positive	3.4E+38	1.7E+308	≅9.2E18	1.1E+4932
Infinities†	Yes	Yes	No	Yes
NaNs†	Yes	Yes	Yes	Yes

†**Denorms** (**denormalized numbers**), **NaNs** (**Not-a-Number**), and **infinities** are defined in Chapter 7.

Usually numbers are stored in a **normalized** form, to afford maximum precision for a given significand width. Maximum precision is achieved if the high order bit in the significand is 1 (that is, $1 \leq$ significand < 2).

Example

In Single, the largest representable number has

significand	=	$2 - 2^{-23}$
	=	$1.11111111111111111111111_2$
exponent	=	127
value	=	$(2 - 2^{-23}) * 2^{127}$
	≅	$3.403 * 10^{38}$

the smallest representable positive normalized number has

significand	=	1
	=	$1.00000000000000000000000_2$
exponent	=	-126
value	=	$1 * 2^{-126}$
	≅	$1.175 * 10^{-38}$

and the smallest representable positive denormalized number (see Chapter 7) has

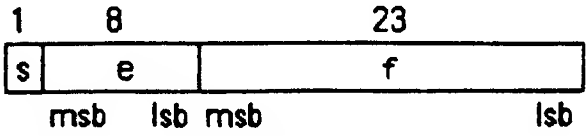
significand	=	2^{-23}
	=	$0.00000000000000000000001_2$
exponent	=	-126
value	=	$2^{-23} * 2^{-126}$
	≅	$1.401 * 10^{-45}$

Formats

This section shows the formats of the four SANE numeric data types. These are pictorial representations and may not reflect the actual byte order in any particular implementation.

Single

A 32-bit single format number is divided into three fields as shown below.



The value v of the number is determined by these fields as follows:

If $0 < e < 255$, then $v = (-1)^s * 2^{(e-127)} * (1.f)$.

If $e = 0$ and $f \neq 0$, then $v = (-1)^s * 2^{(-126)} * (0.f)$.

If $e = 0$ and $f = 0$, then $v = (-1)^s * 0$.

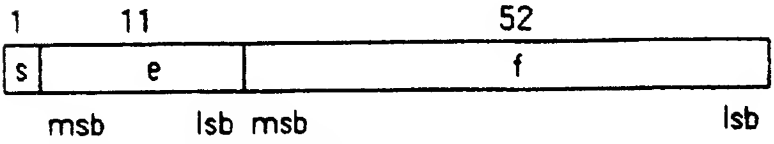
If $e = 255$ and $f = 0$, then $v = (-1)^s * \infty$.

If $e = 255$ and $f \neq 0$, then v is a NaN.

See Chapter 7 for information on the contents of the f field for NaNs.

Double

A 64-bit double format number is divided into three fields as shown below.



The value v of the number is determined by these fields as follows:

If $0 < e < 2047$, then $v = (-1)^s * 2^{(e-1023)} * (1.f)$.

If $e = 0$ and $f \neq 0$, then $v = (-1)^s * 2^{(-1022)} * (0.f)$.

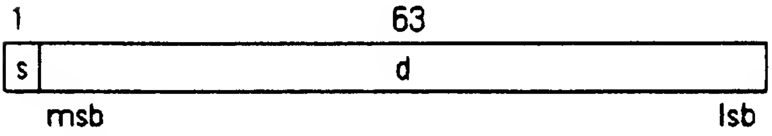
If $e = 0$ and $f = 0$, then $v = (-1)^s * 0$.

If $e = 2047$ and $f = 0$, then $v = (-1)^s * \infty$.

If $e = 2047$ and $f \neq 0$, then v is a NaN.

Comp

A 64-bit comp format number is divided into two fields as shown below.



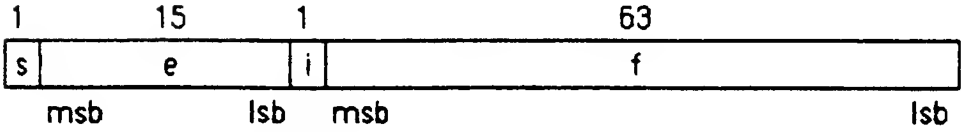
The value v of the number is determined by these fields as follows:

If s = 1 and d = 0, then v is the unique comp NaN.

Otherwise, v is the two's-complement value of the 64-bit representation.

Extended

An 80-bit extended format number is divided into four fields as shown below.



The value v of the number is determined by these fields as follows:

If 0 < = e < 32767, then $v = (-1)^s * 2^{(e-16383)} * (i.f)$.

If e = 32767 and f = 0, then $v = (-1)^s * \infty$, regardless of i.

If e = 32767 and f ≠ 0, then v is a NaN, regardless of i.

Chapter 3

Arithmetic Operations

SANE provides these basic arithmetic operations for the SANE data types:

- add
- subtract
- multiply
- divide
- square root
- remainder
- round to integral value

(See Chapters 9 and 10 for auxiliary operations and higher-level functions supported by SANE.)

All the basic arithmetic operations produce the best possible result: the mathematically exact result coerced to the precision and range of the extended type. The coercions honor the user-selectable rounding direction and handle all exceptions according to the requirements of the IEEE Standard (see Chapter 8).

Remainder

Generally, remainder (and mod) functions are defined by the expression

$x \text{ rem } y = x - y * n$

where n is some integral approximation to the quotient x/y. This expression can be found even in the conventional integer-division algorithm:

$$\begin{array}{lrl} & n & \text{(integral quotient approximation)} \\ \text{(divisor)} & y\,\overline{)\,x\qquad\quad} & \text{(dividend)} \\ & \underline{y * n} & \\ & x - y * n & \text{(remainder)} \end{array}$$

SANE supports the remainder function specified in the IEEE Standard:

> When $y \neq 0$, the remainder $r = x \text{ rem } y$ is defined regardless of the rounding direction by the mathematical relation $r = x - y * n$, where n is the integral value nearest the exact value x/y; whenever $|n - x/y| = 1/2$, n is even. The remainder is always exact. If $r = 0$, its sign is that of x.

Example 1

Find 5 rem 3 . Here $x = 5$ and $y = 3$. Since $1 < 5/3 < 2$ and since $5/3 = 1.66666...$ is closer to 2 than to 1, n is taken to be 2, so

$5 \text{ rem } 3 = r = 5 - 3 * 2 = -1$

Example 2

Find 7.0 rem 0.4 . Since $17 < 7.0/0.4 < 18$ and since $7.0/0.4 = 17.5$ is equally close to both 17 and 18, n is taken to be the even quotient, 18. Hence,

$7.0 \text{ rem } 0.4 = r = 7.0 - 0.4 * 18 = -0.2$

The IEEE remainder function differs from other commonly used remainder and mod functions. It returns a remainder of the smallest possible magnitude, and it always returns an exact remainder. All the other remainder functions can be constructed from the IEEE remainder.

Round to Integral Value

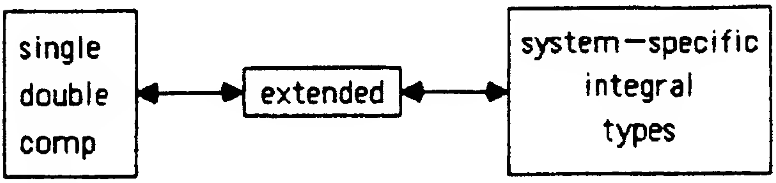
An input argument is rounded according to the current rounding direction to an integral value and delivered to the extended format. For example, 12345678.875 rounds to 12345678.0 or 12345679.0. (The rounding direction, which can be set by the user, is explained fully in Chapter 8.)

Note that, in each floating-point format, all values of sufficiently great magnitude are integral. For example, in single, numbers whose magnitudes are at least 2^{23} are integral.

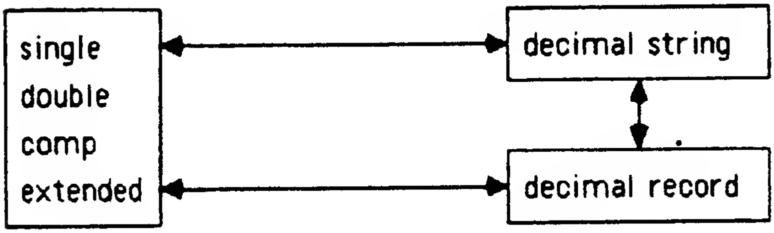
Chapter 4

Conversions

SANE provides conversions between the extended type and each of the other SANE types (single, double, and comp). A particular SANE implementation will provide conversions between extended and those numeric types supported in its particular larger environment. For example, a Pascal implementation will have conversions between extended and the Pascal integer type.



SANE implementations also provide either conversions between decimal strings and SANE types, or conversions between a decimal record type and SANE types, or both. Conversions between decimal records and decimal strings may be included too.



Conversions Between Extended and Single or Double

A conversion to extended is always exact. A conversion from extended to single or double moves a value to a storage type with less range and precision, and sets the overflow, underflow, and inexact exception flags as appropriate. (See Chapter 8 for a discussion of exception flags.)

Conversions to Comp and Other Integral Formats

Conversions to integral formats are done by first rounding to an integral value (honoring the current rounding direction) and then, if possible, delivering this value to the destination format. If the source operand of a conversion from extended to comp is a NaN, an infinity, or out-of-range for the comp format, then the result is the comp NaN and for infinities and values out-of-range, the invalid exception is signaled. If the source operand of a conversion to a system-specific integer type is a NaN, infinity, or out-of-range for that format, then invalid is signaled (unless the type has an appropriate representation for the exceptional result). NaNs, infinities, and out-of-range values are stored in a two's-complement integer format as the extreme negative value (for example, in the 16-bit integer format, as -32768).

Note that IEEE rounding into integral formats differs from most common rounding functions on halfway cases. With the default rounding direction (to nearest), conversions to comp or to a system-specific integer type will round 0.5 to 0, 1.5 to 2, 2.5 to 2, and 3.5 to 4, rounding to even on halfway cases. (Rounding is discussed in detail in Chapter 8.)

Conversions Between Binary and Decimal

The IEEE Standard for binary floating-point arithmetic specifies the set of numerical values representable within each floating-point format. It is important to recognize that binary storage formats can exactly represent the fractional part of decimal numbers in only a few cases; in all other cases, the representation will be approximate. For example, 0.5_{10}, or $1/2_{10}$, can be represented exactly as 0.1_2. On the other hand, 0.1_{10}, or $1/10_{10}$, is a repeating fraction in binary: $0.00011001100...._2$. Its closest representation in single is $0.00011001100110011001100110 1_2$, which is closer to 0.10000000149_{10}than to 0.10000000000_{10}.

As binary storage formats generally provide only close approximations to decimal values, it is important that conversions between the two types be as accurate as possible. Given a rounding direction, for every decimal value there is a best (correctly rounded) binary value for each binary format. Conversely, for any rounding direction, each binary value has a corresponding best decimal representation for a given decimal format. Ideally, binary-decimal conversions should obtain this best value to reduce accumulated errors. Conversion routines in SANE implementations meet or exceed the stringent error bounds specified by the IEEE Standard. This means that although in extreme cases the conversions do not deliver the correctly rounded result, the result delivered is very nearly as good as the correctly rounded result. (See the IEEE Standard [10] for a more detailed description of error bounds.)

Conversions From Decimal Strings to SANE Types

Routines may be provided to convert numeric decimal strings to the SANE data types. These routines are provided for the convenience of those who do not wish to write their own parsers and scanners. Examples of acceptable input are

123 123.4E-12 -123. .456 3e9 -0

-INF Inf NAN(12) -NaN() nan

The 12 in NAN(12) is a NaN code (see Chapter 8).

The accepted syntax is formally defined, using Backus-Naur form, in Table 4-1.

Table 4-1. *Syntax for String Conversions*

<decimal number>	::=	[{space | tab}] <left decimal>
<left decimal>	::=	[+|-] <unsigned decimal>
<unsigned decimal>	::=	<finite number> | <infinity> | <NAN>
<finite number>	::=	<significand> [<exponent>]
<significand>	::=	<integer> | <mixed>
<integer>	::=	<digits> [.]
<digits>	::=	{0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
<mixed>	::=	[<digits>] . <digits>
<exponent>	::=	E [+|-] <digits>
<infinity>	::=	INF
<NAN>	::=	NAN[([<digits>])]

(Note: In the table, square brackets enclose optional items, braces (curly brackets) enclose elements to be repeated at least once, and vertical bars separate alternative elements; letters that appear literally, like the 'E' marking the exponent field, may be either upper or lower case.)

Decform Records and Conversions From SANE Types to Decimal Strings

Each conversion to a decimal string is controlled by a **decform** record, which contains two fields:

style	—	16-bit integer (0 or 1)
digits	—	16-bit integer

Style equals 0 for floating and 1 for fixed. Digits gives the number of significant digits for the floating style and the number of digits to the right of the decimal point for the fixed style (digits may be negative if the style is fixed). Decimal strings resulting from these conversions are always acceptable input for conversions from decimal strings to SANE types. Further formatting details are implementation-dependent.

The Decimal Record Type

The decimal record type provides an intermediate unpacked form for programmers who wish to do their own parsing of numeric input or formatting of numeric output. The decimal record format has three fields:

sgn — 16-bit integer (0 or 1)
exp — 16-bit integer
sig — string (maximum length is implementation-dependent)

The value represented is

$(-1)^{sgn} * sig * 10^{exp}$

when the length of sig is 18 or less. (Some implementations allow additional information in characters past the eighteenth.) Sig contains the integral decimal significand: the initial byte of sig (sig[0]) is the length byte, which gives the length of the ASCII string that is left-justified in the remaining bytes. Sgn is 0 for + and 1 for -. For example, if sgn = 1, exp = -3, and sig = '85' (sig[0] = 2, not shown), then the number represented is -0.085.

Conversions From Decimal Records to SANE Types

Conversions from the decimal record type handle any sig digit-string of length 18 or less (with an implicit decimal point at the right end). The following special cases apply:

- If sig[1] = '0' (zero), the decimal record is converted to zero. For example, a decimal record with sig = '0913' is converted to zero.
- If sig[1] = 'N', the decimal record is converted to a NaN. Except when the destination is of type comp (which has a unique NaN), the succeeding characters of sig are interpreted as a hex representation of the result significand: if fewer than 4 characters follow 'N' then they are right justified in the high-order 15 bits of the field f illustrated in the section "Formats" in Chapter 2; if 4 or more characters follow 'N' then they are left justified in the result's significand; if no characters, or only '0's, follow N, then the result NaN code is set to nanzero = 15 (hex).
- If sig[1] = 'I' and the destination is not of comp type, the decimal record is converted to an infinity. If the destination is of comp type, the decimal record is converted to a NaN and invalid is signaled.
- Other special cases produce undefined results.

Conversions From SANE Types to Decimal Records

Each conversion to a decimal record is controlled by a decform record (see above). All implementations allow at least 18 digits to be returned in sig. The implied decimal point is at the right end of sig, with exp set accordingly.

Zeroes, infinities, and NaNs are converted to decimal records with sig parts '0' (zero), 'I', and strings beginning with 'N', while exp is undefined. For NaNs, 'N' may be followed by a hex representation of the input significand. The third and forth hex digits following 'N' give the NaN code. For example, 'N0021000000000000' has NaN code 21 (hex).

When the number of digits specified in a decform record exceeds an implementation maximum (which is at least 18), the result is undefined.

A number may be too large to represent in a chosen fixed style. For instance, if the implementation's maximum length for sig is 18, then 10^{15} (which requires 16 digits to the left of the point in fixed-style representations) is too large for a fixed-style representation specifying more than 2 digits to the right of the point. If a number is too large for a chosen fixed style, then (depending on the SANE implementation) one of two results is returned: an implementation may return the most significant digits of the number in sig and set exp so that the decimal record contains a valid floating-style representation of the number; alternatively, an implementation may simply set the string sig to '?'. Note that in any implementation, the test

(-exp < > decform digits) or (sig[1] = '?')

determines whether a nonzero finite number is too large for the chosen fixed style.

Conversions Between Decimal Formats

SANE implementations may provide conversions between decimal strings and decimal records.

Conversion From Decimal Strings to Decimal Records

This conversion routine is intended as an aid to programmers doing their own scanning. The routine is designed for use either with fixed strings or with strings being received (interactively) character by character. An integer argument on input gives the starting index into the string, and on output is one greater than the index of the last character in the numeric substring just parsed. The longest possible numeric substring is parsed; if no numeric substring is recognized, then the index remains unchanged. Also, a Boolean argument is returned indicating that the input string, beginning at the input index, is a valid numeric string or a valid prefix of a numeric string. The accepted input for this conversion is the same as for conversions from decimal strings to SANE types (see above). Output is the same as for conversions from SANE types to decimal records (also above).

Examples

Input String	Index In	 Out	Output Value	Valid-Prefix
12	1	3	12	TRUE
12E	1	3	12	TRUE
12E-	1	3	12	TRUE
12E-3	1	6	12E-3	TRUE
12E-x	1	3	12	FALSE
12E-3x	1	6	12E-3	FALSE
x12E-3	2	7	12E-3	TRUE
IN	1	1	UNDEFINED	TRUE
INF	1	4	INF	TRUE

Conversion From Decimal Records to Decimal Strings

This conversion is controlled by the style field of a decform record (the digits field is ignored). Input is the same as for conversions from decimal records to SANE types, and output formatting is the same as for conversions from SANE types to decimal strings. This conversion, actually a formatting operation, is exact and signals no exception.

Chapter 5

Expression Evaluation

SANE arithmetic is extended-based. Arithmetic operations produce results with extended precision and extended range. For minimal loss of accuracy in more complicated computations, you should use extended temporary variables to store intermediate results.

Using Extended Temporaries

A programmer may use extended temporaries deliberately to reduce the effects of round-off error, overflow, and underflow on the final result.

Example 1

To compute the single-precision sum

S = X[1] * Y[1] + X[2] * Y[2] + ... + X[N] * Y[N]

where X and Y are arrays of type single, declare an extended variable XS and compute

```
XS := 0;
FOR I := 1 TO N DO
  XS := XS + X[I] * Y[I];     {extended-precision arithmetic }
S := XS;                      {deliver final result to single.}
```

Even when input and output values have only single precision, it may be very difficult to prove that single-precision arithmetic is sufficient for a given calculation. Using extended-precision arithmetic for intermediate values will often improve the accuracy of single-precision results more than virtuoso algorithms would. Likewise, using the extra range of the extended type for intermediate results may yield correct final results in the single type in cases when using the single type for intermediate results would cause an overflow or a catastrophic underflow. Extended-precision arithmetic is also useful for calculations involving double or comp variables: see Example 2.

Extended-Precision Expression Evaluation

High-level languages that support SANE evaluate all non-integer numeric expressions to extended precision, regardless of the types of the operands.

Example 2

If C is of type comp and MAXCOMP is the largest comp value, then the right-hand side of

C := (MAXCOMP + MAXCOMP) / 2

would be evaluated in extended to the exact result C = MAXCOMP, even though the intermediate result MAXCOMP + MAXCOMP exceeds the largest possible comp value.

Extended-Precision Expression Evaluation and the IEEE Standard

The IEEE Standard encourages extended-precision expression evaluation. Extended evaluation will on rare occasions produce results slightly different from those produced by other IEEE implementations that lack extended evaluation. Thus in a single-only IEEE implementation,

```
z := x + y
```

with x, y, and z all single, is evaluated in one single-precision operation, with at most one rounding error. Under extended evaluation, however, the addition x + y is performed in extended, then the result is coerced to the single precision of z, with at most two rounding errors. Both implementations conform to the standard.

The effect of a single- or double-only IEEE implementation can be obtained under SANE with rounding precision control, as described in Chapter 8.

Chapter 6

Comparisons

SANE supports the usual numeric comparisons: less, less-or-equal, greater, greater-or-equal, equal, and not-equal. For real numbers, these comparisons behave according to the familiar ordering of real numbers.

SANE comparisons handle NaNs and infinities as well as real numbers. The usual trichotomy for real numbers is extended so that, for any SANE values a and b, exactly one of the following is true:

$a < b$
$a > b$
$a = b$
a and b are unordered

Determination is made by the following rule: If x or y is a NaN, then x and y are unordered; otherwise, x and y are less, equal, or greater according to the ordering of the real numbers, with the understanding that $+0 = -0 =$ real 0, and
$-\infty <$ each real number $< +\infty$.

(Note that a NaN always compares unordered—even with itself.)

The meaning of high-level language relational operators is a natural extension of their old meaning based on trichotomy. For example, the Pascal or BASIC expression x < = y is true if x is less than y or if x equals y, and is false if x is greater than y or if x and y are unordered. Note that the SANE not-equal relation means less, greater, or unordered—even if not-equal is written < >, as in Pascal and BASIC. High-level languages supporting SANE supplement the usual comparison operators with a function that takes two numeric arguments and returns the appropriate relation (less, equal, greater, or unordered). This function can be used to determine whether two numeric representations satisfy any combination of less, equal, greater, and unordered.

A high-level language comparison that involves a relational operator containing less or greater, but not unordered, signals invalid if the operands are unordered (that is, if either operand is a NaN). For example, in Pascal or BASIC if x or y is a quiet NaN then x < y, x <= y, x >= y, and x > y signal invalid, but x = y and x <> y (recall that <> contains unordered) do not. If a comparison operand is a signaling NaN, then invalid is always signaled, just as in arithmetic operations.

Chapter 7

Infinities, NaNs, and Denormalized Numbers

In addition to the normalized numbers supported by most floating-point packages, IEEE floating-point arithmetic also supports infinities, NaNs, and denormalized numbers.

Infinities

An **infinity** is a special bit pattern that can arise in one of two ways:

1. When a SANE operation should produce an exact mathematical infinity (such as 1/0), the result is an infinity bit pattern.
2. When a SANE operation attempts to produce a number with magnitude too great for the number's intended floating-point storage format, the result may (depending on the current rounding direction) be an infinity bit pattern.

These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The infinities, one positive (+INF) and one negative (-INF), generally behave as suggested by the theory of limits. For example, 1 added to +INF yields +INF; -1 divided by +0 yields -INF; and 1 divided by -INF yields -0.

Each of the storage types single, double, and extended provides unique representations for +INF and -INF. The comp type has no representations for infinities. (An infinity moved to the comp type becomes the comp NaN.)

NaNs

When a SANE operation cannot produce a meaningful result, the operation delivers a special bit pattern called a **NaN** (Not-a-Number). For example, 0 divided by 0, + INF added to -INF, and sqrt(-1) yield NaNs. A NaN can occur in any of the SANE storage types (single, double, extended, and comp); but, generally, system-specific integer types have no representation for NaNs. NaNs propagate through arithmetic operations. Thus, the result of 3.0 added to a NaN is the same NaN (that is, has the same NaN code). If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: **quiet NaNs**, the usual kind produced by floating-point operations; and **signalling NaNs**.

When a signaling NaN is encountered as an operand of an arithmetic operation, the invalid-operation exception is signaled and, if no halt occurs, a quiet NaN is the delivered result. Signaling NaNs could be used for uninitialized variables. They are not created by any SANE operations. The most significant bit of the field f illustrated in the section "Formats" in Chapter 2 is clear for quiet NaNs and set for signaling NaNs. The unique comp NaN generally behaves like a quiet NaN.

A NaN in a floating-point format has an associated NaN code that indicates the NaN's origin. (These codes are listed in Table 7-1). The NaN code is the 8th through 15th most significant bits of the field f illustrated in Chapter 2. The comp NaN is unique and has no NaN code.

Table 7-1. *SANE NaN Codes*

Name	Dec	Hex	Meaning
NANSQRT	1	$01	Invalid square root, such as sqrt(-1)
NANADD	2	$02	Invalid addition, such as (+INF) - (+INF)
NANDIV	4	$04	Invalid division, such as 0/0
NANMUL	8	$08	Invalid multiplication, such as 0 * INF
NANREM	9	$09	Invalid remainder or mod such as x rem 0
NANASCBIN	17	$11	Attempt to convert invalid ASCII string
NANCOMP	20	$14	Result of converting comp NaN to floating
NANZERO	21	$15	Attempt to create a NaN with a zero code
NANTRIG	33	$21	Invalid argument to trig routine
NANINVTRIG	34	$22	Invalid argument to inverse trig routine
NANLOG	36	$24	Invalid argument to log routine
NANPOWER	37	$25	Invalid argument to x^i or x^y routine
NANFINAN	38	$26	Invalid argument to financial function
NANINIT	255	$FF	Uninitialized storage (signaling NaN)

Denormalized Numbers

Whenever possible, floating-point numbers are **normalized** to keep the leading significand bit 1: this maximizes the resolution of the storage type. When a number is too small for a normalized representation, leading zeros are placed in the significand to produce a **denormalized** representation. A denormalized number is a nonzero number that is not normalized and whose exponent is the minimum exponent for the storage type.

Example

The sequence below shows how a single-precision value becomes progressively denormalized as it is repeatedly divided by 2, with rounding to nearest. This process is called **gradual underflow.**

A_0	$= 1.100\ 1100\ 1100\ 1100\ 1100\ 1101 * 2^{-126} \cong 0.1_{10} * 2^{-122}$
$A_1 = A_0/2$	$= 0.110\ 0110\ 0110\ 0110\ 0110\ 0110 * 2^{-126}$ (underflow)
$A_2 = A_1/2$	$= 0.011\ 0011\ 0011\ 0011\ 0011\ 0011 * 2^{-126}$
$A_3 = A_2/2$	$= 0.001\ 1001\ 1001\ 1001\ 1001\ 1010 * 2^{-126}$ (underflow)
	. . . .
$A_{22} = A_{21}/2$	$= 0.000\ 0000\ 0000\ 0000\ 0000\ 0011 * 2^{-126}$
$A_{23} = A_{22}/2$	$= 0.000\ 0000\ 0000\ 0000\ 0000\ 0010 * 2^{-126}$ (underflow)
$A_{24} = A_{23}/2$	$= 0.000\ 0000\ 0000\ 0000\ 0000\ 0001 * 2^{-126}$
$A_{25} = A_{24}/2$	$= 0.0$ (underflow)

$A_1 \ldots A_{24}$ are denormalized; A_{24} is the smallest positive denormalized number in single type.

Why Denormalized Numbers?

The use of denormalized numbers makes statements like the following true for all real numbers:

$x - y = 0$ if and only if $x = y$

This statement is not true for most older systems of computer arithmetic, because they exclude denormalized numbers. For these systems, the smallest nonzero number is a normalized number with the minimum exponent; when the result of an operation is smaller than this smallest normalized number, the system delivers zero as the result. For such **flush-to-zero** systems, if $x \neq y$ but $x - y$ is smaller than the smallest normalized number, then $x - y = 0$. IEEE systems do not have this defect, as $x - y$, although denormalized, is not zero.

(A few old programs that rely on premature flushing to zero may require modification to work properly under IEEE arithmetic. For example, some programs may test $x - y = 0$ to determine whether x is very near y.)

Inquiries: Class and Sign

Each valid representation in a SANE data type (single, double, comp, or extended) belongs to exactly one of these classes:

- signaling NaN
- quiet NaN
- infinite
- zero
- normalized
- denormalized

SANE implementations provide the user with the facility to determine easily the class and sign of any valid representation.

Chapter 8

Environmental Control

Environmental controls include the rounding direction, rounding precision, exception flags, and halt settings.

Rounding Direction

The available rounding directions are

- to-nearest
- upward
- downward
- toward-zero

The rounding direction affects all conversions and arithmetic operations except comparison and remainder. Except for conversions between binary and decimal (described in Chapter 4), all operations are computed as if with infinite precision and range and then rounded to the destination format according to the current rounding direction. The rounding direction may be interrogated and set by the user.

The default rounding direction is to-nearest. In this direction the representable value nearest to the infinitely precise result is delivered; if the two nearest representable values are equally near, the one with least significant bit zero is delivered. Hence, halfway cases round to even when the destination is the comp or a system-specific integer type, and when the round-to-integer operation is used. If the magnitude of the infinitely precise result exceeds the format's largest value (by at least one half unit in the last place), then the corresponding signed infinity is delivered.

The other rounding directions are upward, downward, and toward-zero. When rounding upward, the result is the format's value (possibly INF) closest to and no less than the infinitely precise result. When rounding downward, the result is the format's value (possibly -INF) closest to and no greater than the infinitely precise result. When rounding toward zero, the result is the format's value closest to and no greater in magnitude than the infinitely precise result. To truncate a number to an integral value, use toward-zero rounding either with conversion into an integer format or with the round-to-integer operation.

Rounding Precision

Normally, SANE arithmetic computations produce results to extended precision and range. To facilitate simulations of arithmetic systems that are not extended-based, the IEEE Standard requires that the user be able to set the rounding precision to single or double. If the SANE user sets rounding precision to single (or double) then all arithmetic operations produce results that are correctly rounded and that overflow or underflow as if the destination were single (or double), even though results are typically delivered to extended formats. Conversions to double and extended formats are affected if rounding precision is set to single, and conversions to extended formats are affected if rounding precision is set to double; conversions to decimal, comp, and system-specific integer types are not affected by the rounding precision. Rounding precision can be interrogated as well as set.

Setting rounding precision to single or double does not significantly enhance performance, and in some SANE implementations may hinder performance.

Exception Flags and Halts

SANE supports five exception flags with corresponding halt settings:

- invalid-operation (or invalid, for short)
- underflow
- overflow

- divide-by-zero
- inexact

These exceptions are signaled when detected; and, if the corresponding halt is enabled, the SANE engine will jump to a user-specified location. (A high-level language need not pass on to its user the facility to set this location, but may halt the user's program). The user's program can examine or set individual exception flags and halts, and can save and get the entire environment (rounding direction, rounding precision, exception flags, and halt settings). Further details of the halt (trap) mechanism are SANE implementation-specific.

Exceptions

The *invalid-operation* exception is signaled if an operand is invalid for the operation to be performed. The result is a quiet NaN, provided the destination format is single, double, extended, or comp. The invalid conditions are these:

- (addition or subtraction) magnitude subtraction of infinities, for example, (+INF) + (-INF);
- (multiplication) 0 * INF;
- (division) 0/0 or INF/INF;
- (remainder) x rem y, where y is zero or x is infinite;
- (square root) if the operand is less than zero;
- (conversion) to the comp format or to a system-specific integer format when excessive magnitude, infinity, or NaN precludes a faithful representation in that format (see Chapter 4 for details);
- (comparison) via predicates involving "<" or ">", but not "unordered", when at least one operand is a NaN;
- any operation on a signaling NaN except sign manipulations (negate, absolute-value, and copy-sign) and class and sign inquiries.

The *underflow* exception is signaled when a floating-point result is both tiny and inexact (and therefore is perhaps significantly less accurate than it would be if the exponent range were unbounded). A result is considered tiny if, before rounding, its magnitude is smaller than its format's smallest positive normalized number.

The *divide-by-zero* exception is signaled when a finite nonzero number is divided by zero. It is also signaled, in the more general case, when an operation on finite operands produces an exact infinite result: for example, logb (0) returns -INF and signals divide-by-zero. (Overflow, rather than divide-by-zero, flags the production of an inexact infinite result.)

The *overflow* exception is signaled when a floating-point destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. (Invalid, rather than overflow, flags the production of an out-of-range value for an integral destination format.)

The *inexact* exception is signaled if the rounded result of an operation is not identical to the mathematical (exact) result. Thus, inexact is always signaled in conjunction with overflow or underflow. Valid operations on infinities are always exact and therefore signal no exceptions. Invalid operations on infinities are described above.

Managing Environmental Settings

The environmental settings in SANE are global and can be explicitly changed by the user. Thus all routines inherit these settings and are capable of changing them. Often special precautions must be taken because a routine requires certain environmental settings, or because a routine's settings are not intended to propagate outside the routine. (Examples in this section use Pascal syntax. SANE implementations in other languages have operations with equivalent functionality.)

Example 1

The subroutine below uses to-nearest rounding while not affecting its caller's rounding direction.

```
           - - -
var r: RoundDir;       { local storage for rounding direction }
           - - -
begin
    r := GetRound;         { save caller's rounding direction }
    SetRound (TONEAREST);  {          set to-nearest rounding }
        - - -
    SetRound (r)       {  restore caller's rounding direction }
end;
```

Note that, if the subroutine is to be reentrant, then storage for the caller's environment must be local.

SANE implementations may provide two efficient functions for managing the environment as a whole: procedure-entry and procedure-exit.

The procedure-entry function returns the current environment (for saving in local storage) and sets the default environment: rounding direction to-nearest, rounding precision extended, and exception flags and halts clear.

Example 2

The following subroutine runs under the default environment while not affecting its caller's environment.

```
          - - -
var e: Environment;             { local storage for environment }
          - - -
begin
    ProcEntry (e);              { save caller's envirnoment and }
                                { set default environment       }
          - - -
    SetEnvironment (e)          { restore caller's environment  }
end;
```

The procedure-exit function facilitates writing subroutines that appear to their callers to be atomic operations (such as addition, sqrt, and others). Atomic operations pass extra information back to their callers by signaling exceptions; however, they hide internal exceptions, which may be irrelevant or misleading. Procedure-exit, which takes a saved environment as arguments, does the following:

1. It temporarily saves the exception flags (raised by the subroutine).
2. It restores the environment received as argument.
3. It signals the temporarily saved exceptions. (Note that if enabled, halts could occur at this step.)

Thus exceptions signaled between procedure-entry and procedure-exit are hidden from the calling program unless the exceptions remain raised when the procedure-exit function is called.

Example 3

The following function signals underflow if its result is denormal, and overflow if its result is infinite, but hides spurious exceptions occurring from internal computations.

```
function compres: double;
           - - -
var e: Environment;            { local storage for environment }
    c: NumClass;               { for class inquiry             }
           - - -
begin {compres}
    ProcEntry (e);             { save caller's environment and }
                               { set default environment -     }
                               { now halts disabled            }
           - - -
    compres := result;                 { result to be returned }
    c := ClassD (result);              { class inquiry         }
    ClearXcps;          { clear possibly spurious exceptions   }

{ now raise specified exception flags:                         }
     if c = INFINITE then SetException (OVERFLDW, TRUE)
     else if c = DENDRMALNUM then SetXcp (UNDERFLDW, TRUE);
     ProcExit (e)             { restore caller's environment,   }
                              { including any halt enables, and }
                              { then signal exceptions from     }
                              { subroutine                      }
end {compres} ;
```

Chapter 9

Auxiliary Procedures

SANE includes a set of special routines that are recommended in an appendix to the IEEE Standard as aids to programming:

- negate
- absolute value
- copy-sign
- next-after
- scalb
- logb

Sign Manipulation

The sign manipulation operations change only the sign of their argument. Negate reverses the sign of its argument. Absolute-value makes the sign of its argument positive. Copy-sign takes two arguments and copies the sign of one of its arguments onto the sign of its other argument.

These operations are treated as nonarithmetic in the sense that they raise no exceptions: even signaling NaNs do not signal the invalid-operation exception.

Next-After Functions

The floating-point values representable in single, double, and extended formats constitute a finite set of real numbers. The next-after functions (one for each of these formats) generate the next representable neighbor in the proper format, given an initial value x and another value y indicating a direction from the initial value.

Each of the next-after functions takes two arguments, x and y:

nextsingle(x,y)	(x and y are single)
nextdouble(x,y)	(x and y are double)
nextextended(x,y)	(x and y are extended)

As elsewhere, the names of the functions may vary with the implementation.

Special Cases for Next-After Functions

If the initial value and the direction value are equal, then the result is the initial value.

If the initial value is finite but the next representable number is infinite, then overflow and inexact are signaled.

If the next representable number lies strictly between -M and +M, where M is the smallest positive normalized number for that format, and if the arguments are not equal, then underflow and inexact are signaled.

Binary Scale and Log Functions

The scalb and logb functions are provided for manipulating binary exponents.

Scalb efficiently scales a given number (x) by a given integer power (n) of 2, returning x * 2^n.

Logb returns the binary exponent of its input argument as a signed integral value. When the input argument is denormalized, the exponent is determined as if the input argument had first been normalized.

Special Cases for Logb

If x is infinite, logb(x) returns +INF.

If x = 0, logb(x) returns -INF and signals divide-by-zero.

Chapter 10

Elementary Functions

SANE provides a number of basic mathematical functions, including logarithms, exponentials, two important financial functions, trigonometric functions, and a random number generator. These functions are computed using the basic SANE arithmetic heretofore described.

All of the elementary functions, except the random number generator, handle NaNs, overflow, and underflow appropriately. All signal inexact appropriately, except that the general exponential and the financial functions may conservatively signal inexact when determining exactness would be too costly.

Logarithm Functions

SANE provides three logarithm functions:

base-2 logarithm	:	$\log_2(x)$
base-e or natural logarithm	:	$\ln(x)$
base-e logarithm of 1 plus argument	:	ln1(x)

Ln1(x) accurately computes $\ln(1 + x)$. If the input argument x is small, such as an interest rate, the computation of ln1(x) is more accurate than the straightforward computation of $\ln(1 + x)$ by adding x to 1 and taking the natural logarithm of the result.

Special Cases for Logarithm Functions

If x = +INF, then $\log_2(x)$, ln(x), and ln1(x) return +INF. No exception is signaled.

If x = 0, then $\log_2(x)$ and ln(x) return -INF and signal divide-by-zero. Similarly, if x = -1, then ln1(x) returns -INF and signals divide-by-zero.

If x < 0, then $\log_2(x)$ and ln(x) return a NaN and signal invalid. Similarly, if x < -1, then ln1(x) returns a NaN and signals invalid.

Exponential Functions

SANE provides five exponential functions:

base-2 exponential	:	2^x
base-e or natural exponential	:	e^x
base-e exponential minus 1	:	exp1(x)
integer exponential	:	x^i (i of integer type)
general exponential	:	x^y

Exp1(x) accurately computes e^x - 1. If the input argument x is small, such as an interest rate, then the computation of exp1(x) is more accurate than the straightforward computation of e^x - 1 by exponentiation and subtraction.

Special Cases for 2^x, e^x, exp1(x)

If x = +INF, then 2^x, e^x, and exp1(x) return +INF. No exception is signaled.

If x = -INF, then 2^x and e^x return 0; and exp1(x) returns -1. No exception is signaled.

Special Cases for x^i

If the integer exponent i equals 0 and x is not a NaN, then x^i returns 1. Note that with the integer exponential, $x^0 = 1$ even if x is zero or infinite.

If x is +0 and i is negative, then x^i returns +INF and signals divide-by-zero.

If x is -0 and i is negative, then x^i returns +INF if i is even, or -INF if i is odd: both cases signal divide-by-zero.

Special Cases for x^y

If x is +0 and y is negative, then the general exponential x^y returns +INF and signals divide-by-zero.

If x is -0 and y is integral and negative, then x^y returns +INF if y is even, or -INF if y is odd: both cases signal divide-by-zero.

The general exponential x^y returns a NaN and signals invalid if

- both x and y equal 0;
- x is infinite and y equals 0;
- x = 1 and y is infinite; or
- x is -0 or less than 0 and y is nonintegral.

Financial Functions

SANE provides two functions, compound and annuity, that can be used to solve various financial, or time-value-of-money, problems.

Compound

The compound function computes

$$\text{compound}(r,n) = (1 + r)^n$$

where r is the interest rate and n is the number (perhaps nonintegral) of periods. When the rate r is small, compound gives a more accurate computation than does the straightforward computation of $(1 + r)^n$ by addition and exponentiation.

Compound is directly applicable to computation of present and future values:

$$PV = FV * (1 + r)^{(-n)} = \frac{PV}{\text{compound}(r,n)}$$

$$FV = PV * (1 + r)^n = PV * \text{compound}(r,n)$$

Annuity

The annuity function computes

$$\text{annuity}(r,n) = \frac{1 - (1 + r)^{(-n)}}{r}$$

where r is the interest rate and n is the number of periods. Annuity is more accurate than the straightforward computation of the expression above using basic arithmetic operations and exponentiation. The annuity function is directly applicable to the computation of present and future values of ordinary annuities:

$$PV = PMT * \frac{1 - (1 + r)^{(-n)}}{r}$$

$$= PMT * \text{annuity}(r,n)$$

$$FV = PMT * \frac{(1 + r)^{n} - 1}{r}$$

$$= PMT * (1 + r)^{n} * \frac{1 - (1 + r)^{(-n)}}{r}$$

$$= PMT * \text{compound}(r,n) * \text{annuity}(r,n)$$

where PMT is the amount of one periodic payment.

Special Cases for compound(r,n)

If r = 0 and n is infinite, or if r = -1, then compound(r,n) returns a NaN and signals invalid.

If r = -1 and n < 0, then compound(r,n) returns +INF and signals divide-by-zero.

Special Cases for annuity(r,n)

If r = 0, then annuity(r,n) computes the sum of 1 + 1 + ... + 1 over n periods, and therefore returns the value n and signals no exceptions (the value n corresponds to the limit as r approaches 0).

If r < -1, then annuity(r,n) returns a NaN and signals invalid.

If r = -1 and n > 0, then annuity(r,n) returns -INF and signals divide-by-zero.

Trigonometric Functions

SANE provides the basic trigonometric functions

cosine	:	cos(x)
sine	:	sin(x)
tangent	:	tan(x)
arctangent	:	arctan(x)

The arguments for cosine, sine, and tangent and the results of arctangent are expressed in radians. The cosine, sine, and tangent functions use an argument reduction based on the remainder function (see Chapter 3) and the nearest extended-precision approximation of pi/2. Thus the cosine, sine, and tangent functions have periods slightly different from their mathematical counterparts and diverge from their counterparts when their arguments become large. Number results from arctangent lie between -pi/2 and pi/2.

The remaining trigonometric functions can be easily and efficiently computed from the elementary functions provideo (see Appendix A).

Special Cases for sin(x), cos(x)

If x is infinite, then cos(x) and sin(x) return a NaN and signal invalid.

Special Cases for tan(x)

If x is the nearest extended approximation to ±pi/2, then tan(x) returns ±INF.

If x is infinite, then tan(x) returns a NaN and signals invalid.

Special Case for arctan(x)

If x = ±INF, then arctan(x) returns the nearest extended approximation to ±pi/2.

Random Number Generator

SANE provides a pseudorandom number generator, random. Random has one argument, passed by address. A sequence of (pseudo) random integral values r in the range

$1 \leq r \leq 2^{31} - 2$

can be generated by initializing an extended variable r to an integral value (the seed) in the above range and making repeated calls random(r); each call delivers in r the next random number in the sequence.

If seed values of r are nonintegral or outside the range

$1 \leq r \leq 2^{31} - 2$

then results are unspecified.

A pseudorandom rectangular distribution on the interval (0,1) can be obtained by dividing the results from random by

$2^{31} - 1$ = scalb (31,1) - 1 .

Appendix A

Other Elementary Functions

The Standard Apple Numeric Environment (SANE) provides several transcendental functions; from these, you can construct other high-quality functions, as shown by the pseudocode examples below. These robust, accurate functions are based on algorithms developed by Professor William Kahan of the University of California at Berkeley.

All variables in the pseudocode below are extended. The constant C is 2^{-33} = scalb (-33,1). C is chosen to be nearly the largest value for which 1 - C^2 rounds to 1.

Exception Handling

Unlike the SANE elementary functions, these functions do not provide complete handling of special cases and exceptions. The most troublesome exceptions can be correctly handled if you

- begin each function with a call to procedure-entry;
- clear the spurious exceptions indicated in the comment;
- end each function with a call to procedure-exit (see Chapter 8).

Functions

Secant

```
sec(x) ← 1 / cos(x)
```

CoSecant

```
csc(x) ← 1 / sin(x)
```

CoTangent

```
cot(x) ← 1 / tan(x)
```

ArcSine

```
y ← |x|
If y ≥ 0.3 then begin
                     y ← Atan (x/sqrt ((1 - x) * (1 + x)))
                     {spurious divide-by-zero may arise}
                   end
else if y ≥ C then y ← Atan (x / (sqrt (1 - x^2))
                 else y ← x
arcsin(x) ← y
```

ArcCosine

```
arccos(x) ← 2 * Atan (sqrt ((1 - x)/(1 + x)) )
{spurious divide-by-zero may arise}
```

Sinh

```
y ← |x|
If y ≥ C then begin
                   y ← exp1(y)
                   y ← 0.5 * (y + y/(1 + y))
                 end
copy the sign of x onto y
sinh(x) ← y
```

Cosh

```
y ← exp(|x|)
cosh(x) ← 0.5 * y + 0.25 / (0.5 * y)
```

Tanh

```
y ← |x|
If y ≥ C then begin
                 y ← exp1(-2 * y)
                 y ← -y/(2 + y)
              end
copy the sign of x onto y
tanh(x) ← y
```

ArcSinh

```
y ← |x|
If y ≥ C then begin
                y ← ln1 (y + y / (1/y + sqrt(1 + (1/y)^2) ))
                {spurious underflow may arise}
             end
copy the sign of x onto y
asinh(x) ← y
```

ArcCosh

```
y ← |x|
acosh(x) ← ln1 ( (sqrt (y-1)) * (sqrt (y-1) + sqrt (y+1)) )
```

ArcTanh

```
y ← |x|
If y ≥ C then y ← ln1 (2 * y/(1 - y)) / 2
copy the sign of x onto y
atanh(x) ← y
```

Glossary

Application type: A data type used to store data for applications.

Arithmetic type: A data type used to hold results of calculations inside the computer. The SANE arithmetic type, extended, has greater range and precision than the application types, in order to improve the mathematical properties of the application types.

Binary floating-point number: A string of bits representing a sign, an exponent, and a significand. Its numerical value, if any, is the signed product of the significand and two raised to the power of its exponent.

Comp type: A 64-bit application data type for storing integral values of up to 18- or 19-decimal-digit precision. It is used for accounting applications, among others.

Decform record: A data type for specifying the formatting for decimal results (of conversions). It specifies fixed- or floating-point form and the number of digits.

Denormalized number, or **denorm:** A nonzero binary floating-point number that is not normalized (that is, whose significand has a leading bit of zero) and whose exponent is the minimum exponent for the number's storage type.

Double type: A 64-bit application data type for storing floating-point values of up to 15- or 16-decimal-digit precision. It is used for statistical and financial applications, among others.

Environmental settings: The rounding direction and rounding precision, plus the exception flags and their respective halts.

Exceptions: Special cases, specified by the IEEE Standard, in arithmetic operations. The exceptions are invalid, underflow, overflow, divide-by-zero, and inexact.

Exceptlon flag: Each exception has a flag that can be set, cleared and tested. It is set when its respective exception occurs and stays set until explicitly cleared.

Exponent: The part of a binary floating-point number that indicates the power to which two is raised in determining the value of the number. The wider the exponent field in a numeric type, the greater range it will handle.

Extended type: An 80-bit arithmetic data type for storing floating-point values of up to 19- or 20-decimal-digit precision. SANE uses it to hold the results of arithmetic operations.

Flush-to-zero: A system that excludes denormalized numbers. Results smaller than the smallest normalized number are rounded to zero.

Gradual underflow: A system that includes denormalized numbers.

Halt: Each exception has a halt-enable that can be set or cleared. When an exception is signaled and the corresponding halt is enabled, the SANE engine will transfer control to the address in a halt vector. A high-level language need not pass an to its user the facility to set the halt vector, but may halt the user's program. Halts remain set until explicitly cleared.

Infinity: A special bit pattern produced when a floating-point operation attempts to produce a number greater in magnitude than the largest representable number in a given format. Infinities are signed.

Integer types: System types for integral values. Integer types typically use 16- or 32-bit two's complement integers. Integer types are not SANE types but are available to SANE users.

Integral value: A value in a SANE type that is exactly equal to a mathematical integer: ..., -2, -1, 0, 1, 2, ...

NaN (Not a Number): A special bit pattern produced when a floating-point operation cannot produce a meaningful result (for example, 0/0 produces a NaN). NaNs can also be used for uninitialized storage. NaNs propagate through arithmetic operations.

Normallzed number: A binary floating-point number in which all significand bits are significant: that is, the leading bit of the significand is 1.

Quiet NaN: A NaN that propagates through arithmetic operations without signaling an exception (and hence without halting a program).

Rounding direction: When the result of an arithmetic operation cannot be represented exactly in a SANE type, the computer must decide how to round the result. Under SANE, the computer resolves rounding decisions in one of four directions, chosen by the user: tonearest (the default), upward, downward, and towardzero.

Sign bit: The bit of a single, double, comp, or extended number that indicates the number's sign: 0 indicates a positive number; 1, a negative number.

Signaling NaN: A NaN that signals an invalid exception when the NaN is an operand of an arithmetic operation. If no halt occurs, a quiet NaN is produced for the result. No SANE operation creates signaling NaNs.

Significand: The part of a binary floating-point number that indicates where the number falls between two successive powers of two. The wider the significand field in a numeric type, the more resolution it will have.

Single type: A 32-bit application data type for storing floating-point values of up to 7- or 8-decimal-digit precision. It is used for engineering applications, among others.

Annotated Bibliography

[1] *Apple III Pascal Programmer's Manual*, Volume 2. "Appendix A: The TRANSCEND and REALMODES Units" and "Appendix E: Floating-Point Arithmetic." Cupertino, Calif.: Apple Computer, Inc., 1981.

These appendixes describe the implementation of single-precision arithmetic in Apple III Pascal, which was based upon Draft 8.0 of the proposed Standard.

[2] *Apple III Pascal Numerics Manual: A Guide to Using the Apple III Pascal SANE and Elems Units*. Cupertino, Calif.: Apple Computer, Inc., 1983.

This manual describes the Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units. This was Apple's first full implementation of IEEE arithmetic.

[3] *Apple Pascal Numerics Manual: A Guide to Using the Apple Pascal SANE and Elems Units*. Cupertino, Calif.: Apple Computer, Inc., 1983.

This manual, generalized from [2], describes the Apple II and Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units.

[4] Cody, W. J. "Analysis of Proposals for the Floating-Point Standard." *IEEE Computer* Vol. 14, No. 3 (March 1981).

This paper compares the several contending proposals presented to the Working Group.

[5] Coonen, Jerome T.. "Accurate, Yet Economical Binary-Decimal Conversions." To appear in *ACM Transactions on Mathematical Software*.

[6] Coonen, Jerome T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic." *IEEE Computer* Vol. 13, No. 1 (January 1980).

This paper is a forerunner to the work on the draft Standard.

[7] Coonen, Jerome T. "Underflow and the Denormalized Numbers." *IEEE Computer* Vol. 14, No. 3 (March 1981).

[8] Demmel, James. "The Effects of Underflow on Numerical Computation." To appear in *SIAM Journal on Scientific and Statistical Computing*.

These papers examine one of the major features of the proposed Standard, gradual underflow, and show how problems of bounded exponent range can be handled through the use of denormalized values.

[9] Fateman, Richard J. "High-Level Language Implications of the Proposed IEEE Floating-Point Standard." *ACM Transactions on Programming Languages and Systems* Vol. 4, No. 2 (April 1982).

This paper describes the significance to high-level languages, especially FORTRAN, of various features of the IEEE proposed Standard.

[10] Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Standard for Binary Floating-Point Arithmetic." Proposed to IEEE, 345 East 47th Street, New York, NY 10017.

The implementation of SANE is based upon the final draft of this Standard, submitted December 1982.

[11] Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Proposed Standard for Binary Floating-Point Arithmetic." *IEEE Computer* Vol. 14, No. 3 (March 1981).

This is Draft 8.0 of the proposed Standard, which was offered for public comment. The current Draft 10.0 is substantially simpler than this draft; for instance, warning mode and projective mode have been eliminated, and the definition of underflow has changed. However, the intent of the Standard is basically the same, and this paper includes some excellent introductory comments by David Stevenson, Chairman of the Floating-Point Working Group.

[12] Hough, D. "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic." *IEEE Computer* Vol. 14, No. 3 (March 1981).

This paper is an excellent introduction to the floating-point environment provided by the proposed Standard, showing how it facilitates the implementation of robust numerical computations.

[13] Kahan, W. "Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard." In *Interval Mathematics 1980*, edited by K. E. L. Nickel. New York: Academic Press, 1980.

This paper shows how the proposed Standard facilitates interval arithmetic.

[14] Kahan, W., and Jerome T. Coonen. "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments." In *The Relationship between Numerical Computation and Programming Languages*, edited by J. K. Reid. New York: North Holland, 1982.

This paper describes high-level language issues relating to the proposed IEEE Standard, including expression evaluation and environment handling.

Part II: The 6502 Assembly-Language SANE Engine

Table of Contents

Chapter 1

Introduction

The purpose of the software package described in Part II of this manual is to provide the features of the Standard Apple Numeric Environment (SANE) to assembly-language programmers using Apple's 6502-based systems. SANE—described in detail in Part I of this manual—fully supports the IEEE Standard (754) for Binary Floating-Point Arithmetic, and augments the Standard to provide greater utility for applications in accounting, finance, science, and engineering. The IEEE Standard and SANE offer a combination of quality, predictability, and portability heretofore unknown for numerical software.

A functionally equivalent 68000 assembly-language SANE engine is available for Apple's 68000-based systems. Thus numerical algorithms coded in assembly language for an Apple 6502-based system can be readily recoded for an Apple 68000-based system. We have chosen macros for accessing the 6502 and 68000 engines to make it easier to port algorithms from one systen to the other. The 68000 SANE engine is described in Part III.

Part II of this manual describes the use of the 6502 assembly-language SANE engine, but does not describe SANE itself. For example, Part II explains how to call the SANE remainder function from 6502 assembly language but does not discuss what this function does. See Part I for information about the semantics of SANE.

See Appendix A for information about which version of the 6502 assembly-language SANE engine is appropriate for your Apple 6502-based system (Apple II series, 128K Apple II series, or Apple III series) and how to install that version in your system.

Chapter 2

Basics

The following code illustrates a typical invocation of the SANE engine, FP6502.

```
PUSH     A_ADR     ; push address of A (single format)
PUSH     B_ADR     ; push address of B (extended format)
FSUBS              ; Floating-point SUBtract Single: B ← B - A
```

PUSH and FSUBS are assembly-language macros taken from the macro file listed in Appendix B. The form of the operation in the example (B ← B - A, where A is a numeric type and B is extended) is similar to the forms for most FP6502 operations. Also, this example is typical of SANE engine calls because operands are usually passed to FP6502 by pushing the addresses of the operands onto the stack prior to the call. Details of SANE engine access are given later in this chapter.

The SANE elementary functions are provided in Elems6502. Access to Elems6502 is similar to access to FP6502; details are given in Chapter 9.

Operation Forms

The example above illustrates the form of an FP6502 binary operation. Forms for other FP6502 operations are described in this section. Examples and further details are given in subsequent chapters.

Arithmetic and Auxiliary Operations

Most numeric operations are either unary (one operand), like square root and negation, or binary (two operands), like addition and multiplication.

The 6502 assembly-language SANE engine, FP6502, provides unary operations in a one-address form:

DST ← <op> DST ... for example, B ← sqrt(B)

The operation <op> is applied to (or operates on) the operand DST and the result is returned to DST, overwriting the previous value. DST is called the destination operand.

FP6502 provides binary operations in a two-address form:

DST ← DST <op> SRC ... for example, B ← B / A

The operation <op> is applied to the operands DST and SRC and the result is returned to DST, overwriting the previous value. SRC is called the source operand.

In order to store the result of an operation (unary or binary), the location of the operand DST must be known to FP6502, so DST is passed by address to FP6502. In general all operands, both source and destination, are passed by address to FP6502. The only exceptions are operands in the 16-bit integer format, which for certain operations are passed by value.

For most operations the storage format for a source operand (SRC) can be the 16-bit integer format or one of the SANE numeric formats (single, double, extended, or comp). To support the extended-based SANE arithmetic, a destination operand (DST) must be in the extended format.

The next-after functions have the two-address form

DST ← DST <op> SRC

but differ from the conventions above in that SRC and DST are both single, both double, or both extended.

Conversions

FP6502 provides conversions between the extended format and other SANE formats, between extended and 16-bit integers, and between extended and decimal records. Conversions between binary formats (single, double, extended, comp, and integer) and conversions from decimal to binary have the form

DST ← SRC

Conversions from binary to decimal have the form

DST ← SRC according to SRC2

where SRC2 is a decform record specifying the decimal format for the conversion of SRC to DST.

Comparisons

Comparisons have the form

<relation> ← SRC, DST

where DST is extended and SRC is single, double, comp, extended, or integer, and where <relation> is less, equal, greater, or unordered according as

SRC <relation> DST

Here the result <relation> is returned in the 6502's registers, rather than in a memory location.

Other Operations

FP6502 provides inquiries for determining the class and sign of an operand and operations for accessing the floating-point environment word and the halt address. Forms for these operations vary and will be given as the operations are introduced.

External Access

The SANE engine, FP6502, is accessed through a JSR to the beginning of the engine memory image. We call this entry point FP6502.

A program using FP6502 should initialize the environment word. If the program enables halts, then the halt vector must be properly initialized. (See Chapters 7 and 8 regarding the environment word and the halt vector.)

FP6502 uses 52 bytes of the zero page for temporary storage. (See Appendix A for details.) FP6502 does not preserve the pre-call contents of these 52 bytes. Note that FP6502's use of the zero page is temporary, and hence the user need not preserve the contents of the 52 bytes of the zero page between calls to FP6502.

The A, X, and Y registers and CPU status flags are not preserved by the engine. Some operations return information in the X and Y registers and in the status flags. Other operations leave the A, X, and Y registers and status flags unspecified.

When called, the engine removes its input arguments from the stack and returns no results on the stack. Temporary stack growth during engine calls does not exceed 20 bytes. On exit, decimal mode is clear.

The access constraints described in this section also apply to Elems6502.

Calling Sequence

A typical invocation of the engine consists of a sequence of 6502 assembly-language instructions and macros

```
PUSH    <source (address)>
PUSH    <destination address>
PUSH    <opword>
JSR     FP6502
```

where PUSH pushes the high byte, then the low byte:

```
LDA     %1+1
PHA
LDA     %1
PHA
```

Other calls may have more or fewer operands to push onto the stack. All source operands are pushed before the destination operand.

The Opword

The opword contains an operand format code in its high-order byte and an operation code in its low-order byte.

The operand format code specifies the format (extended, double, single, integer, or comp) of one of the operands. The operand format code typically gives the format for the source operand (SRC). At most one operand format need be specified, because other operands are always extended (or, in the case of next-after, because both operands are of the same format).

The operation code specifies the operation to be performed by FP6502.

(Opwords are listed in Appendix C; operand format codes and operation codes are listed in Appendix B.)

Example

The format code for single is 0200 (hex). The operation code for divide is 0006 (hex). Hence the opword 0206 (hex) indicates divide by a value of type single.

Assembly-Language Macros

The sample macros (see Appendix B) combine PUSH <opword> and JSR FP6502 to provide mnemonics for calls to the 6502 SANE engine.

Example 1

Add a single-format operand A to an extended-format operand B.

```
PUSH    A_ADR    ; push address of A
PUSH    B_ADR    ; push address of B
FADDS            ; Floating-point ADD Single: B ← B + A
```

Example 2

Compute B ← sqrt(A), where A and B are extended. The value of A should be preserved.

```
PUSH    A_ADR    ; push address of A
PUSH    B_ADR    ; push address of B
FX2X             ; Floating-point eXtended to eXtended: B ← A
PUSH    B_ADR    ; push address of B
FSQRTX           ; Floating SQuare RooT eXtended: B ← sqrt(B)
```

Example 3

Compute C ← A - B, where A, B, and C are in the double format. Because destinations are extended, a temporary extended variable T is required.

```
PUSH    A_ADR    ; push address of A
PUSH    T_ADR    ; push address of 10 byte temporary variable
FD2X             ; Fl-pt convert Double to eXtended: T ← A
PUSH    B_ADR    ; push address of B
PUSH    T_ADR    ; push address of temporary
FSUBD            ; Fl-pt SUBtract Double: T ← T - B
PUSH    T_ADR    ; push address of temporary
PUSH    C_ADR    ; push address of C
FX2D             ; Fl-pt convert eXtended to Double: C ← T
```

Arithmetic Abuse

FP6502 is designed to be as robust as possible, but it is not bullet-proof. Passing the wrong number of operands to the engine damages the stack. Using UNDEFINED opword parameters or passing incorrect addresses produces undefined results.

Chapter 3

Data Types

FP6502 fully supports the SANE data types

single	—	32-bit floating-point
double	—	64-bit floating-point
comp	—	64-bit integer
extended	—	80-bit floating-point

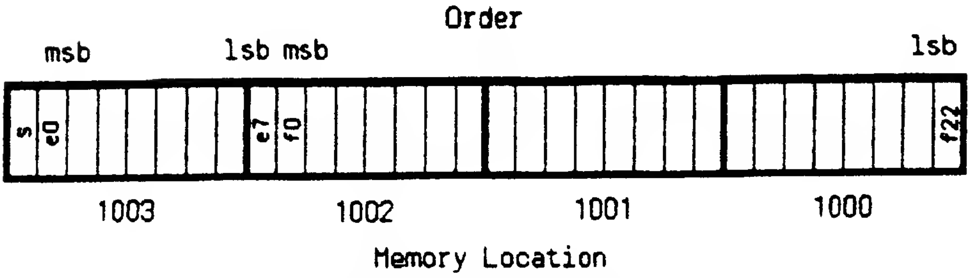
and the 6502-specific type

integer	—	16-bit two's complement integer

The 6502 engine uses the convention that least-significant bytes are stored in low memory. For example, let us take a variable of type single with bits

s	—	sign
e0 ... e7	—	exponent (msb...lsb)
f0 ... f22	—	significand fraction (msb...lsb)

The logical structure of this four-byte variable is shown below:



If this variable is assigned the address 1000, then its bits are distributed to the locations 1000 to 1003 as shown.

The other SANE formats (see Chapter 2 in Part I) are represented in memory in similar fashion.

Chapter 4

Arithmetic Operations and Auxiliary Routines

The operations covered in this chapter follow the access schemes described in Chapter 2.

Unary operations follow the one-address form:
DST ← <op> DST. They use the calling sequence

```
PUSH    <DST address>
PUSH    <opword>
JSR     FP6502
```

Binary operations follow the two-address form:
DST ← DST <op> SRC. They use the calling sequence

```
PUSH    <SRC (address)>
PUSH    <DST address>
PUSH    <opword>
JSR     FP6502
```

The destination operand (DST) for these operations is passed by address and is usually in the extended format. Generally, the source operand (SRC) is passed by address and may be single, double, comp, extended, or (16-bit) integer. Some operations are distinguished by passing the source operand by value, by requiring some specific type for SRC, by using a non-extended destination, or by returning auxiliary information in the X and Y registers and in the processor status bits. In this section, operations so distinguished are noted. The examples employ the macros in Appendix B.

Add, Subtract, Multiply, and Divide

These are binary operations and follow the two-address form.

Example

B ← B / A , where A is double and B is extended.

```
PUSH    A_ADR   ; push address of A
PUSH    B_ADR   ; push address of B
FDIVD           ; divide with source operand of type double
```

Square Root

This is a unary operation and follows the one-address form.

Example

B ← sqrt(B) , where B is extended.

```
PUSH    B_ADR   ; push address of B
FSQRTX          ; square root (operand is always extended)
```

Round-to-Integer, Truncate-to-Integer

These are unary operations and follow the one-address form.

Round-to-integer rounds (according to the current rounding direction) to an integral value in the extended format. Truncate-to-integer rounds toward zero (regardless of the current rounding direction) to an integral value in the extended format. The calling sequence is the usual one for unary operations, illustrated above for square root.

Remainder

This is a binary operation and follows the two-address form.

Remainder returns auxiliary information. The seven low-order bits of the magnitude of the integer quotient n are returned in the X register. The N status bit is set if and only if n is negative. The Y register receives 80 (hex) if n is negative and 0 otherwise.

Example

B ← B rem A , where A is single and B is extended.

```
PUSH    A_ADR    ; push address of A
PUSH    B_ADR    ; push address of B
FREMS            ; remainder with source operand of type single
```

Logb, Scalb

Logb is a unary operation and follows the one-address form.

Scalb is a binary operation and follows the two-address form. Its form is unusual in that its source operand is a 16-bit integer passed by value.

Example

$B \leftarrow B * 2^{130}$, where B is extended.

```
LDA      #0        ; push high byte
PHA                ;      of source
LDA      #82       ; push low byte of source
PHA                ;      (82 hex = 130 decimal)
PUSH     B_ADR     ; push address of B
FSCALBX            ; scalb
```

Negate, Absolute Value, Copy-Sign

Negate and absolute value are unary operations and follow the one-address form. Copy-sign is a binary operation and follows the two-address form.

Example

Copy the sign of a comp A into the sign of an extended B.

```
PUSH     A_ADR     ; push address of A
PUSH     B_ADR     ; push address of B
FCPYSGNC           ; copy-sign with source of type comp
```

Next-After

The next-after operations are binary and use the two-address form; they require both source and destination operands to be of the same floating-point type (single, double, or extended).

Example

B ← next-after(B) in the direction of A, where A and B are double (so *next-after* means *next-double-after*).

```
PUSH     A_ADR     ; push address of A
PUSH     B_ADR     ; push address of B
FNEXTD             ; next-after in double format
```

Chapter 5

Conversions

This chapter discusses conversions between binary formats and conversions between binary and decimal formats.

Conversions Between Binary Formats

FP6502 provides conversions between the extended type and the SANE types single, double, and comp, as well as the 16-bit integer type.

Conversions to Extended

FP6502 provides conversions of a source, of type single, double, comp, extended, or integer, to an extended destination.

		single
		double
extended	←	comp
		extended
		integer

All operands, even integer ones, are passed by address. The following example illustrates the calling sequence.

Example

Convert A to B, where A is of type comp and B is extended.

```
PUSH    A_ADR    ; push address of A
PUSH    B_ADR    ; push address of B
FC2X             ; convert comp to extended
```

Conversions From Extended

FP6502 provides conversions of an extended source to a destination of type single, double, comp, extended, or integer.

single
double
comp ← extended
extended
integer

(Note that conversion to a narrower format may alter values.) Contrary to the usual scheme, the destination for these conversions need not be of type extended. All operands are passed by address. The following example illustrates the calling sequence.

Example

Convert A to B where A is extended and B is double.

```
PUSH    A_ADR    ; push address of A
PUSH    B_ADR    ; push address of B
FX2D             ; convert extended to double
```

Binary-Decimal Conversions

FP6502 provides conversions between the binary types (single, double, comp, extended, and integer) and the decimal record type.

Decimal records and decform records (used to specify the form of decimal representations) are described in Chapter 4 of Part I. For FP6502, the maximum length of the sig digits field of a decimal record is 28. (The value 28 is specific to this implementation: algorithms intended to port to other SANE implementations should use no more than 18 digits in sig.) The integer fields of decimal and decform records conform to the 6502 convention of storing the least significant byte in low memory.

Binary to Decimal

The calling sequence for a conversion from a binary format to a decimal record passes the address of a decform record, the address of a binary source operand, and the address of a decimal-record destination.

Example

Convert a comp-format value A to a decimal record D according to the decform record F.

```
PUSH    F_ADR    : push address of F
PUSH    A_ADR    ; push address of A
PUSH    D_ADR    ; push address of D
FC2DEC           ; convert comp to decimal
```

Fixed-Format "Overflow"

If a number is too large for a chosen fixed style, then FP6502 returns the 28 most significant digits of the number in the sig field of the decimal record and sets the exp field so that the decimal record contains a valid floating-point representation of the number. (Other SANE implementations may simply set sig to the string '?'.)

Decimal to Binary

The calling sequence for a conversion from decimal to binary passes the address of a decimal-record source operand and the address of a binary destination operand.

Example

Convert the decimal record D to a double-format value B.

```
PUSH    D_ADR    ; push address of D
PUSH    B_ADR    ; push address of B
FDEC2D           ; convert decimal to double
```

Techniques for Maximum Accuracy

The following technique applies to FP6502; other SANE implementations require other techniques.

If you are writing a parser and must handle a number with more than 28 significant digits, follow these rules:

1. Place the implicit decimal point to the right of the 28 most significant digits.
2. If any of the discarded digits to the right of the implicit decimal point are nonzero, then
   - signal the inexact exception, and
   - if (a) the number is positive and the rounding direction is upward or if (b) the number is negative and the rounding direction is downward, then take the successor of the last (28th) ASCII character to guarantee a correctly rounded result. (The successor of '9' is ':'.)

Chapter 6

Comparisons and Inquiries

Comparisons

FP6502 offers two comparison operations: FCPX (which signals invalid if its operands compare unordered) and FCMP (which does not). Each compares a source operand (which may be single, double, comp, extended, or 16-bit integer) with a destination operand (which must be extended). The result of a comparison is the relation (less, greater, equal, or unordered) for which

SRC <relation> DST

is true. The result is delivered in the Z, N, and V status bits and redundantly in the X and Y registers:

Result	**Status Bits**			**X Register**	**Y Register**
	Z	**N**	**V**		
greater	0	0	1	40	40
less	0	1	0	80	80
equal	1	0	0	02	00
unordered	0	0	0	01	01

(Register values are given in hex.) Note that the X and Y registers hold the same value unless the relation is equal: this is a byproduct of an implementation optimization.

The IEEE Standard specifies that a relational operator that involves less or greater but not unordered should signal invalid if the operands are unordered. These relational operators are implemented by choosing the comparison that signals invalid appropriately.

Example 1

Test A < = B, where A is single and B is extended; if TRUE branch to LOC; signal if unordered.

```
PUSH     A_ADR     ; push address of A
PUSH     B_ADR     ; push address of B
FCPXS              ; compare using source of type single,
                   ; signal invalid if unordered
FBLE     LOC       ; branch if A <= B
```

Example 2

Test A not-equal B, where A is double and B is extended; if TRUE branch to LOC. (Note that not-equal is equivalent to less, greater, or unordered, so invalid should not be signaled on unordered.)

```
PUSH     A_ADR     ; push address of A
PUSH     B_ADR     ; push address of B
FCMPD              ; compare using source of type double,
                   ; do not signal invalid if unordered
FBNE     LOC       ; branch if A not-equal B
```

Inquiries

The classify operation provides both class and sign inquiries. This operation takes one source operand (single, double, extended, comp, or integer), which is passed by address.

The class of the operand is returned in the X register as

FC	—	signaling NaN
FD	—	quiet NaN
FE	—	infinite
FF	—	zero
00	—	normal
01	—	denormal

Note that as 8-bit two's complement representations, these values are -4, -3, -2, -1, 0, and 1.

The N status bit receives the sign bit of the operand. Redundantly, the Y register is set to 80 (hex) if the sign bit is set and is set to 0 otherwise.

Example

Branch to LOC if the single-format value A is an infinity.

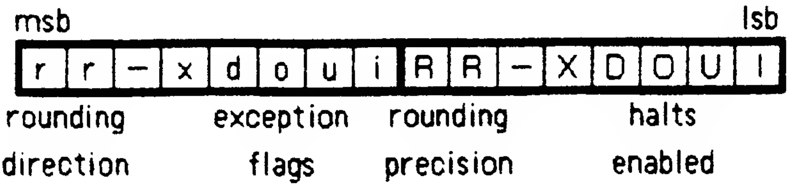
```
PUSH     A_ADR     ; push address of A
FCLASSS            ; classify single
FBINF    LOC       ; branch on infinite to LOC
```

Chapter 7

Environmental Control

The Environment Word

The floating-point environment is encoded in the 16-bit integer format as shown below in hexadecimal:



rounding direction, bits C000			rr
0000	—	to-nearest	
4000	—	upward	
8000	—	downward	
C000	—	toward-zero	
exception flags, bits 1F00			
0100	—	invalid	i
0200	—	underflow	u
0400	—	overflow	o
0800	—	division-by-zero	d
1000	—	inexact	x

rounding precision, bits 00C0			RR
0000	—	extended	
0040	—	double	
0080	—	single	
00C0	—	UNDEFINED	

halts enabled, bits 001F			
0001	—	invalid	I
0002	—	underflow	U
0004	—	overflow	O
0008	—	division-by-zero	D
0010	—	inexact	X

Bits 2000 and 0020 are not used.

Note that the default environment is represented by the integer value zero.

Example

With rounding toward-zero, inexact and underflow exception flags raised, extended rounding precision, and halt on invalid, overflow, and division-by-zero, the most significant byte of the environment is D2 and the least significant byte is 0D.

You gain access to the environment through the operations get-environment, set-environment, test-exception, set-exception, procedure-entry, and procedure-exit.

Get-Environment and Set-Environment

Get-environment takes no input operand. The least significant and most significant bytes of the environment are returned in the X and Y registers, respectively.

Set-environment has one input operand: a 16-bit integer, passed by value, that is interpreted as an environment word.

Example

Set rounding direction to downward.

```
FGETENV             ; get environment
TYA                 ; A  ←  msbyte
AND       #03F      ; clear CO bits
DRA       #080      ; set bits for round downward
PHA                 ; push msbyte
TXA                 ; A  ←  lsbyte
PHA                 ; push lsbyte
FSETENV             ; set environment
```

Test-Exception and Set-Exception

Test-exception has one integer operand, passed by value, which is regarded as a sum of the hex values

01	—	invalid
02	—	underflow
04	—	overflow
08	—	divide-by-zero
10	—	inexact

If an exception flag is set for any of the corresponding bits set in the operand, then test-exception clears the Z flag in the 6502 status register; otherwise, test-exception sets the Z flag.

Example

Branch to XLOC if invalid or overflow is set.

```
LDA      #0        ; A  ←  0
PHA                ; push msbyte
LDA      #5        ; A  ←  'invalid' + 'overflow'
PHA                ; push lsbyte
FTESTXCP           ; test exception
BNE      XLDC      ; branch if Z is clear
```

Set-exception takes one integer operand, passed by value, which encodes a set of exceptions in the manner described above for test-exception. Set-exception stimulates the exceptions indicated in the operand.

Procedure-Entry and Procedure-Exit

Procedure-entry saves the current floating-point environment (16-bit integer) at the address passed as the sole operand, and sets the operative environment to the default state.

Procedure-exit saves (temporarily) the exception flags, sets the environment passed by value as the sole operand, and then stimulates the saved exceptions.

Example

Here is a procedure that appears to its callers as an atomic operation.

```
ATOMICPROC
        PUSH    E_AOR    ; push address to store environment
        FPROCENTRY       ; procedure entry

        ...body of routine...

        PUSH    E        ; push saved environment
        FPROCEXIT        ; procedure exit
        RTS
E       .WORO            ; storage for saved environment
E_AOR   .WORO   E        ; address of E
```

Chapter 8

Halts

FP6502 lets you transfer program control when selected floating-point exceptions occur. Because this facility will be used to implement halts in high-level languages, we refer to it as a halt mechanism. The assembly-language programmer can write a *halt handler* routine to cause special actions for floating-point exceptions. The FP6502 halting mechanism differs from the traps that are an optional part of the IEEE Standard.

Conditions for a Halt

Any floating-point exception will, if the corresponding halt is enabled, trigger a halt. The halt for a particular exception is enabled when two conditions are met:

- The halt (trap) vector, which can be set using the operation set-halt-vector, is not zero.
- The halt-enable bit corresponding to that exception is set.

The Halt Mechanism

If the halt for a given exception is enabled, FP6502 does these things when that exception occurs:

1. FP6502 returns the same result to the destination address that it would return if the halt were not enabled. (However, FP6502 does not set floating-point exception flags for the current operations and does not return results to the X, Y, and P registers.)
2. It leaves the caller's return address on the top-of-stack.

3. It sets the X and Y registers to the least and most significant bytes, respectively, of the address of a record containing status information.
4. It jumps to the address given by the halt vector.

The status information record that the X and Y registers point to has the following fields:

halt vector	—	1 word
environment (state)	—	1 word
result info	—	1 word
opword of user call to FP6502	—	1 word
exceptions from current operation	—	1 byte

"Exceptions from current operation" are encoded in the manner of the exception flags contained in the most significant byte of the floating-point environment (see Chapter 7). The least significant and most significant bytes of "result info" contain what the X and Y registers would have contained on normal (halts not enabled) exit from FP6502. "Result info" can be used by a halt handler routine to set the relevant bits of the 6502 status byte, as well as the X and Y registers, to their normal exit state. (The example below illustrates this use of "result info".)

Set-halt-vector has one input operand: a 16-bit integer, passed by value, that is interpreted as the halt vector (that is, the address to jump to in case a halt occurs).

Get-halt-vector takes no input operand and returns the least and most significant bytes of the halt vector in the X and Y registers, respectively.

Using the Halt Mechanism

This example illustrates the use of the halt mechanism. The user must set the halt vector to the starting address of a halt handler routine. This particular halt handler returns control to the user's program (directly following the call to FP6502 that caused the halt) as if no halt occurred.

```
; Set halt (trap) vector.
        PUSH    HHAOR   ; push address of halt handler
        FSETTV          ; set halt vector
        - - -
        F???            ; call to FP6502 causing halt
        ???             ; halt handler returns control to here
        - - -
HH      ; halt handler routine starts here
; Store address in X and Y registers into temporary location.
        STX     TEMP
        STY     TEMP+1  ; temp points to status info record
; OR exceptions from current operation into environment.
        LOY     #8
        LDA     (TEMP),Y        ; A ← current exceptions
        LOY     #3
        ORA     (TEMP),Y        ; A ← A OR msbyte of environment
        STA     (TEMP),Y        ; msbyte of environment ← A
; Operate on result info to set registers and status bits as
; though no halt occurred.
        INY                     ; Y ← 4
        LOA     (TEMP),Y
        TAX                     ; X ← lsbyte of result info
        INY                     ; Y ← 5
        LDA     (TEMP),Y
        STA     TEMP            ; TEMP ← msbyte of result info
        BIT     TEMP            ; determines V bit
        TAY                     ; Y ← msbyte of result info
                                ; determines N and Z bits
; Return to user operation after call to FP6502 which triggered
; halt (address already on top-of-stack).
        RTS
        - - -
HHADR   .WORD   HH      ; HHAOR contains address of HH
```

Chapter 9

Elementary Functions

The elementary functions that are specified by the Standard Apple Numeric Environment are made available to the 6502 assembly-language programmer in Elems6502. Also included are two functions that compute $\log_2(1 + x)$ and $2^x - 1$ accurately. Elems6502 calls the SANE engine (FP6502) for its basic arithmetic. The access schemes for FP6502 (described in Chapter 2) and Elems6502 are similar. Opwords and sample macros are included at the end of the file SANEMACRO.TEXT listed in Appendix B. (These macros are used freely in the examples below.)

One-Argument Functions

The SANE elementary functions $\log_2(x)$, $\ln(x)$, $\ln1(x) = \ln(1 + x)$, 2^x, e^x, $\exp1(x) = e^x - 1$, cos(x), sin(x), tan(x), atan(x), and random(x), together with $\log21(x) = \log_2(1 + x)$ and $\exp21(x) = 2^x - 1$, each have one extended argument, passed by address. These functions use the one-address calling sequence

```
PUSH    DST address
PUSH    <opword>
JSR     ELEMS6502
```

to effect

DST ← <op> DST

This calling sequence follows the FP6502 access scheme for unary operations, such as square root and negate.

Example

B ← sin(B), where B is of extended type.

```
PUSH    B_ADR    ; push address of B
FSINX            ; B ← sin(B)
```

Two-Argument Functions

General exponentiation (x^y) has two extended arguments, both passed by address. The result is returned in x.

The function uses the calling sequence for binary operations

```
PUSH    SRC address     ; push exponent address first
PUSH    DST address     ; push base address second
PUSH    <opword>
JSR     ELEMS6502
```

to effect

$$DST \leftarrow DST^{SRC}$$

Integer exponentiation (x^i) has two arguments. The extended argument x, passed by address, receives the result. The 16-bit integer argument i is passed by value.

The function uses the modified calling sequence for binary operations

```
PUSH    SRC value       ; push integer exponent value first
PUSH    DST address     ; push base address second
PUSH    <opword>
JSR     ELEMS6502
```

to effect

$$DST \leftarrow DST^{SRC}$$

Example

$B \leftarrow B^3$, where the type of B is extended.

```
LDA      #0        ; A ← 0
PHA                ; push msbyte of source
LDA      #3        ; A ← 3
PHA                ; push lsbyte of source
PUSH     B_ADR     ; push address of B
FXPWRI             ; integer exponentiation
```

Three-Argument Functions

Compound and annuity use the calling sequence

```
PUSH     SRC2 address     ; push address of rate first
PUSH     SRC address      ; push address of number of periods second
PUSH     DST  address     ; push address of destination third
PUSH     <opword>
JSR      ELEMS6502
```

to effect

DST ← <op> (SRC2, SRC)

where <op> is compound or annuity, SRC2 is the rate, and SRC is the number of periods. All arguments SRC2, SRC, and DST must be of the extended type.

Example

$C \leftarrow (1 + R)^N$, where C, R, and N are of type extended.

```
PUSH     R_ADR     ; push address of R
PUSH     N_ADR     ; push address of N
PUSH     C_ADR     ; push address of C
FCOMPOUND          ; compound
```

Appendix A

6502 SANE: Installation and Access

Introduction

This appendix discusses implementations of SANE (FP6502 and Elems6502) for three families of machines: the generic Apple II, the 128K Apple II, and the Apple III. These three implementations differ slightly, as do the methods required for the assembly-language programmer to use them. For each machine family, this appendix discusses assembly-language programming both within and outside the Pascal environment.

The Product Disks

You should have three disks: SANE1 in ProDOS™(SOS) format, and SANE2 and SANE3 in Apple II Pascal format.

The SANE3 Disk

The SANE3 disk contains the FP6502 and Elems6502 files needed for assembly-language programming with the Pascal Assembler. If you are using the Pascal Assembler, then the SANE3 disk is the only disk you will normally need and you can skip the descriptions of the SANE2 disk, the SANE1 disk, and the customizing process.

The SANE3 disk also permits you to use an Apple III to produce customized versions of FP6502 and Elems6502 primarily for use outside the Pascal system, for example with the ProDOS Assembler Tools. The customization procedures are described below in "Customizing Files for Use With ProDOS and DOS."

SANE3 contains the following files:

A3.CUSTOM.CODE	—	the customizing program described below
A3.CUSTOM.LIB	—	library used by the customizing program
CUSTOMIZE.DATA	—	data file used by the customizing program
A2X.FPINIT.CODE	—	Pascal unit for loading A2X.AFP.CODE
SANEMACRO.TEXT	—	macros for use with the Pascal Assembler

It contains these code files for the Pascal Assembler:

A2.AFP.CODE	—	FP6502 for Apple II
A3.AFP.CODE	—	FP6502 for Apple III
A2X.AFP.CODE	—	FP6502 for 128K Apple II (not relocatable)
A2.AELEM.CODE	—	Elems6502 for Apple II
A3.AELEM.CODE	—	Elems6502 for Apple III
A2X.AELEM.CODE	—	Elems6502 for 128K Apple II

The SANE2 Disk

The SANE2 disk permits an Apple II to produce customized versions of FP6502 and Elems6502, primarily for use outside the Pascal system, for example with the ProDOS Assembler Tools. SANE2 is a startup disk that contains code files used in the customization process:

A2.AFP.CODE

A3.AFP.CODE

A2.AELEM.CODE

A3.AELEM.CODE

A2X.AELEM.CODE

CUSTOMIZE.DATA

It also contains the Pascal run-time system:

SYSTEM.APPLE

SYSTEM.PASCAL

SYSTEM.MISCINFO

SYSTEM.CHARSET

SYSTEM.LIBRARY — library support for the customizing program

SYSTEM.STARTUP — the customizing program itself

The SANE1 Disk

The SANE1 disk is in ProDOS (SOS) format. It contains these Pascal ASCII files (BASIC text files):

A2X.BANKSW
A2X.LOADER — Self-documenting source files for the ProDOS Assembler. These files are discussed below with the 128K Apple II.

INCLUDE.EQUS — Definitions of the constants that appear in the macro files. To use the macros INCLUDE.EQUS must be included in your source file.

GENERIC.MACROS — Sample SANE macros in no particular assembler format. Before use, you must edit them to suit your assembler.

PDOS.SANEMACRO/ — A subdirectory containing SANE macros (one per file) for the ProDOS Assembler.

Customizing Files for Use With ProDOS and DOS

You can use SANE2 and SANE3 to produce custom versions of FP6502 and Elems6502, primarily for use outside the Pascal system. Both disks can produce custom files in either Apple II Pascal or ProDOS (SOS) format for use on either the Apple II or the Apple III.

To customize using an Apple II (64K or larger), use SANE2. To customize using an Apple III, use SANE3. Before you begin, prepare a disk in either ProDOS (SOS) format or Apple II Pascal format to receive the transferred files.

If you have an Apple II, start up with the SANE2 disk and follow the directions.

If you have an Apple III, start up with your own Pascal, X(ecute /SANE3/A3.CUSTOM and follow the directions.

The program lets you specify

- The target machine (Apple II or Apple III) that will ultimately run your versions of FP6502 and Elems6502
- Relocatable code files (for use with the Pascal Assembler) or absolute code files
- Starting addresses (ORG) for absolute code files
- Zero-page locations used by FP6502 and Elems6502 (These locations overlap.)

Before the customizer reads or writes any files, it shows you a summary of your choices. At any stage, you can press ESCAPE to return to the main menu and start over.

The customizer retains the usual Pascal header and trailer blocks when it produces relocatable code files, but the absolute code files it creates do not contain these extra blocks.

▲ **Warning**
FP6502 keeps an internal copy of the current floating-point environment (the rounding direction, exception flags, rounding precision, and halt settings). When FP6502 is used with the Pascal system, it is possible to construct executable code that contains two copies of FP6502. When this happens, your code may set the environment in one copy. This environment may appear to be lost when subsequent code accesses a different copy of FP6502.

To avoid such anomalies,

- *Arrange files so that FP6502 is involved only ONCE in the linking process. If you link more than once to FP6502, each linked code segment will contain its own copy of FP6502.*
- *Do not access the Pascal unit SANE and also link FP6502 into your assembly language in the same program. If you do, one copy of FP6502 will be loaded with the Pascal Interface and another will be linked into your assembly-language code.*

Except for a waste of space, there is no danger in having two copies of Elems6502 in your code.

Apple II: 64K or 128K

This section describes versions of FP6502 and Elems6502 that can be used on any Apple II containing 64K or more memory.

Assembly-Language Access: Pascal

This section describes the use of FP6502 and Elems6502 with the Apple II Pascal Assembler.

Files

The files A2.AFP.CODE (FP6502) and A2.AELEM.CODE (Elems6502) are on the SANE3 disk. You need not customize these files unless you want to alter zero-page usage.

Addressing

Assembly-language programs that call FP6502 (Elems6502) must include the directive(s)

```
.REF FP6502
.REF ELEMS6502     (if Elems6502 is required).
```

Linking

Programs accessing FP6502 or Elems6502 from assembly language must consist of

a	Pascal host program
calling	assembly-language subroutines
which in turn call	FP6502 and Elems6502

and possibly other code files. To produce executable code, link

the	host program
to	assembly-language subroutines
to	(other code files)
to	A2.AFP.CODE
to	A2.AELEM.CODE (if Elems6502 is required).

For information about using the linker, see *Apple III Pascal Program Preparation Tools*, Chapter 3, or the *Apple II Apple Pascal Operating System Reference Manual*, Chapter 7.

Zero-Page

FP6502 and Elems6502 as supplied use locations 00-33 (hex) as temporary scratch space. Anything stored in these locations is likely to be destroyed by calls to FP6502 or Elems6502. The contents of these locations may be arbitrary whenever FP6502 or Elems6502 is called.

Assembly-Language Access: ProDOS or DOS

This section describes the use of FP6502 and Elems6502 with ProDOS or DOS.

Files

Use the customizing process described above to produce absolute code versions of FP6502 and Elems6502 with starting addresses and zero-page usage of your choice.

Addressing

FP6502 and Elems6502 each have one entry point. Assembly-language programs call them with the instructions

```
JSR       FP6502
JSR       ELEMS6502
```

Calling programs must include the directives

```
FP6502      EQU   <customized starting address>
ELEMS6502   EQU   <customized starting address>
```

Zero-Page

The 52 zero-page locations used by FP6502 and Elems6502 are specified by the user when customizing. These locations are used as temporary scratch space and their contents may be arbitrary whenever FP6502 or Elems6502 is called.

Loading

The procedure for loading FP6502 or ELEMS6502 into memory depends on which operating system you are using. With DOS or ProDOS, just BLOAD to the starting addresses you specified when you customized your files.

Apple II: 128K

A special version of FP6502, A2X.AFP.CODE, is available to the 128K user. It resides in the top 8K of the auxiliary language card, saving 6K of user memory and offering more programming flexibility. It is somewhat harder to use than the generic version of FP6502. The remainder of this section describes this special version.

Assembly-Language Access: Pascal

This section describes the use of the special auxiliary-language-card version of FP6502 and its companion version of Elems6502 with the Apple II Pascal Assembler.

Files

The files A2X.AFP.CODE (FP6502) and A2X.AELEM.CODE (Elems6502) are on the SANE3 disk. You need not customize these files unless you want to alter zero-page usage.

Addressing

Assembly-language programs that call the auxiliary language card version of FP6502 (Elems6502) must include the directive(s)

```
FP6502    .EQU      0E000
.REF ELEMS6502
```

(if Elems6502 is required).

Loading

There are several ways to transfer the FP6502 code to its home on the auxiliary language card:

- Put A2X.AFP.CODE on the prefix volume or on the root volume, install the FPINIT unit in your SYSTEM.LIBRARY (use the library program to transfer A2X.FPINIT.CODE), and X(ecute the following program:

  ```
  program load;
  uses fpinit;
  begin
  end.
  ```

  The program is effective because the Pascal Unit FPINIT has initialization code that transfers A2X.AFP.CODE from either the prefix or root volume to its home on the auxiliary language card and initializes the alternate stack pointer.

- Put A2X.AFP.CODE on either the prefix or the root volume, install the FPINIT unit in your SYSTEM.LIBRARY, and put the declaration 'USES FPINIT' in your host Pascal program. As explained above, FPINIT transfers A2X.AFP.CODE to the auxiliary language card. This solution costs you a small amount of memory space (less than 1/4 K) because the FPINIT code becomes part of your program.

- If you have the special auxiliary language card version of the Pascal unit SANE then you can use this unit instead of FPINIT. Both contain the same loading code.

Linking

Programs accessing FP6502 or Elems6502 from assembly language must consist of

a	Pascal host program
calling	assembly-language subroutines
which in turn call	FP6502 and Elems6502

and possibly other code files. To produce executable code, link

the	host program
to	assembly-language subroutines
to	(other code files)
to	A2X.AELEM.CODE (if Elems6502 is required).

For information about using the linker, see *Apple III Pascal Program Preparation Tools*, Chapter 3, or the *Apple II Apple Pascal Operating System Reference Manual*, Chapter 7.

Zero-Page

FP6502 and Elems6502 as supplied use locations $80-B3 (hex) on the auxiliary zero-page as temporary scratch space. Anything stored in these locations is likely to be destroyed by calls to FP6502 or Elems6502. The contents of these locations may be arbitrary whenever FP6502 or Elems6502 is called.

Switching to Auxiliary Stack, Zero-Page, Language Card

The auxiliary language card versions of both FP6502 and Elems6502 *require* that arguments be passed on the auxiliary stack. There are four basic steps involved in switching:

1. Save the CPU status register, so the current state of interrupts can be restored later.
2. Disable interrupts during the switching operation.
3. Exchange the stack pointer you have been using for the auxiliary (or main) pointer.
4. Restore the CPU status register to its original state.

These commented listings outline recommended ways to accomplish this.

```
ALTZP      .EQU    0C009   ;soft switches.  See the Apple II
MAINZP     .EQU    0C008   ;  Reference Manual for details.
FP6502     .EQU    0E000

.REF        ELEMS6502

STATUS     .BYTE          ;these normally come at the end
TEMPX      .BYTE          ;  so they will not appear in the
TEMPY      .BYTE          ;  position of executable code.

-----------------------------------------------------------------

; Standard protocol to switch in the alt lang card, stack, zero-page

MAIN_2_ALT   PHP                ;save status register on MAIN STACK!
             SEI                ;disable interrupts, stack ptr bad
             PLA                ;pick up status from stack
             STA      STATUS    ;  and put in local storage for later
             STA      ALTZP     ;switch to alt lang card, zero-page
             TSX                ;pick up main stack pointer
             STX      0100      ;  deposit it in standard place
             LDX      0101      ;resurrect alt stack pointer
             TXS                ;  place into 6502
             LDA      STATUS    ;collect original status register
             PHA                ;  put on the AUX STACK
             PLP                ;  and then into P-register

-----------------------------------------------------------------
```

The switch to auxiliary is complete. Now put arguments and opcode on the (auxiliary) stack and call either FP6502 or Elems6502.

```
------------------------------------------------------------------------
; Switch back to main lang card, stack, zero-page.
;   Note that the X, Y registers are saved.
;   The order of their restoration is important
;   because it also conditions the Status register.
;   Results that FP6502 returns in X, Y, P are not lost.

  ALT_2_MAIN  STX     TEMPX    ;hold a copy of X to prevent trashing
              STY     TEMPY    ;  ditto, Y
              SEI              ;disable interrupts: stack ptr change
              TSX              ;pick up alt stack pointer
              STX     0101     ;  keep a copy of it
              LDX     0100     ;load main stack pointer from storage
              STA     MAINZP   ;switch to main lang card, zero-page
              TXS              ;  and restore stack ptr to 6502
              LDA     STATUS   ;collect original status
              PHA              ;  and put it on the stack
              PLP              ;  then put it in the CPU
------------------------------------------------------------------------
```

```
------------------------------------------------------------------------
; Now restore X, Y, and Status registers

  REG_RESTORE LDX     TEMPX
              BIT     TEMPY    ;conditions P register
              LDY     TEMPY    ;also conditions P register
------------------------------------------------------------------------
```

The calling sequence is complete. This detour to the auxiliary language card is transparent to the Pascal system. Note that much of your assembly code might be very happy to run in the auxiliary stack and zero-page. You can probably keep the number of switches to a minimum.

Assembly-Language Access: ProDOS or DOS

This section describes the use of the special auxiliary-language-card version of FP6502 and its companion version of Elems6502 with ProDOS or DOS.

Files

Use the version of FP6502 called A2X.AFP.CODE. You need not customize FP6502 unless you want to alter its zero-page usage. If you do customize, the address of FP6502 should remain at $E000 and the resulting file will no longer have a 512-byte header.

Use the customizing process described above to produce an absolute code version of Elems6502. When customizing Elems, you will be asked to report the starting address of FP6502. Enter `E000`. You will also be asked to select zero-page usage. To make Elems6502 usage consistent with the supplied version of FP6502, answer `80` for zero-page indirect and `88` for zero-page scratch.

Addressing

FP6502 and Elems6502 each have one entry point. Assembly-language programs call them with the instructions

```
JSR        FP6502
JSR        ELEMS6502
```

(if Elems6502 is required).

Calling programs must include the directives

```
FP6502      EQU      $E000
ELEMS6502   EQU      <customized starting address>
```

Zero-Page

During customization you specify the zero-page locations used by Elems6502. You also change the FP6502 usage of $80-B3 to make a new version of FP6502 (still starting at $E000). These locations are used as temporary scratch space and their contents may be arbitrary whenever FP6502 or Elems6502 is called. Note that both files use *auxiliary* zero-page.

Loading

The procedure for loading the disk image of Elems6502 into memory depends on which operating system you are using. With DOS or ProDOS, just BLOAD the file to the starting address specified when you customized.

You can load FP6502 from Pascal as described in the section "Assembly-Language Access: Pascal." If Pascal is not available, then load FP6502 in the way described in /SANE1/A2X.LOADER.

Location of Data

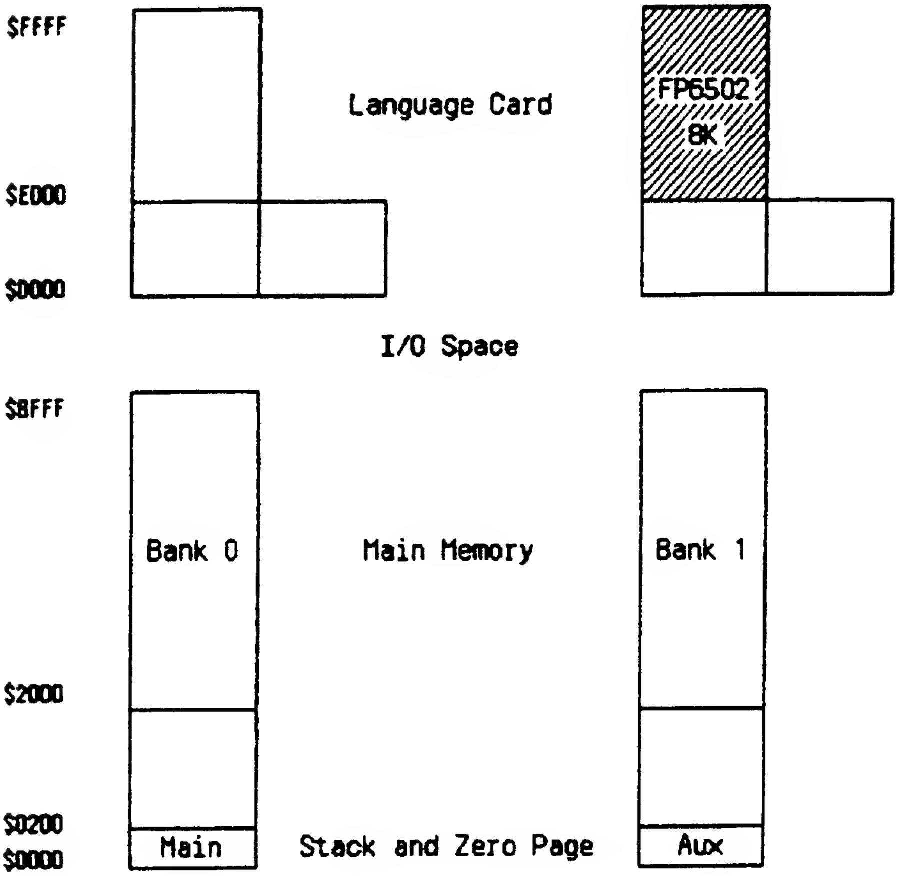
If your programs are very large, or use a huge data space, you might need to use both main and auxiliary RAM banks. If the program and its data must reside in different banks, then the method described in /SANE1/A2X.BANKSW will allow FP6502 access to both the program and its data. Note that you cannot use such a scheme for Elems6502 because Elems6502 itself must reside in either the main or auxiliary RAM bank.

Switching to Auxiliary Stack, Zero-Page, Language Card

The auxiliary language card versions of FP6502 and Elems6502 **require** that arguments be passed on the auxiliary stack. The necessary steps are discussed in the section "Assembly-Language Access: Pascal."

Details of the 128K Implementation

Here is a diagram of the location of the 128K version of FP6502.



There are two significant advantages to using this version of FP6502:

- The auxiliary language card space shown in the diagram above is reserved by Apple Computer, Inc. for FP6502. It should not be used for any other purpose. The version of FP6502 that occupies this space saves you 6K of usable memory. (Elems6502 is not relocated to the auxiliary language card.)
- This is the only version of FP6502 that permits access to the floating-point engine from both assembly language and the Pascal language in the same program. This dual access requires that you use the special auxiliary-language-card version of the Pascal unit SANE. This unit and your assembly-language code both call the 128K version of FP6502. By contrast see the warning box in the introduction to this appendix.

The disadvantage to this scheme is an awkwardness of access. The auxiliary language card is relatively hard to use. When it is switched in for use, the auxiliary stack and zero-page come with it. This shuts off the main stack, main zero-page, and main language card. Therefore to use this version of FP6502, you must also use the auxiliary stack and zero-page.

Note: Once FP6502 is located in the auxiliary language card, it will not be lost unless you turn the computer off, or unless you write over it. Restarting, switching to ProDOS or BASIC, and running a program will not dislodge FP6502 unless the program uses Apple's reserved space on the auxiliary language card.

▲ **Warning**

If you are using

- *an Apple IIe with Apple IIe Extended 80-Column Text Card,*
- *the 128K version of FP6502, and*
- *a device (such as a mouse) that might cause interrupts during floating point arithmetic calls*

then interrupts occurring while FP6502 is active will not be handled correctly by old Apple IIe ROMs. To see whether you have old Apple IIe ROMs, type the following line in Applesoft BASIC:

```
IF (PEEK(-1101)=6) AND (PEEK(-1088)=234)
   THEN PRINT "OLD APPLE //E ROMS"
```

Your dealer can help you upgrade these ROMs.

Apple III

This section describes versions of FP6502 and Elems6502 that can be used on any Apple III.

Assembly-Language Access: Pascal

This section describes the use of FP6502 and Elems6502 with the Apple III Pascal Assembler.

Files

The files A3.AFP.CODE (FP6502) and A3.AELEM.CODE (Elems6502) are on the SANE3 disk. You need not customize these files unless you want to alter zero-page usage.

Addressing

Assembly-language programs that call FP6502 (Elems6502) must include the directive(s)

```
.REF FP6502
.REF ELEMS6502
```
(if Elems6502 is required).

Linking

Programs accessing FP6502 or Elems6502 from assembly language must consist of

a Pascal host program
calling assembly-language subroutines
which in turn call FP6502 and Elems6502

and possibly other code files. To produce executable code, link

the host program
to assembly-language subroutines
to (other code files)
to A3.AFP.CODE
to A3.AELEM.CODE (if Elems6502 is required).

For information about using the linker, see *Apple III Pascal Program Preparation Tools*, Chapter 3, or the *Apple II Apple Pascal Operating System Reference Manual*, Chapter 7.

Zero-Page

The supplied versions of FP6502 and Elems6502 use locations 00-2B (hex) as temporary scratch space, and locations E0-E7(hex) for enhanced indirect addressing. Anything stored in these locations is likely to be destroyed by calls to FP6502 or Elems6502. The contents of these locations may be arbitrary whenever FP6502 or Elems6502 is called.

Location of Data

Although its CPU can address only 64K bytes at one time, the Apple III can handle more memory through bank switching and enhanced indirect addressing. Your linked assembly-language program is placed in one bank pair. Pascal programs and data might be located in other bank pairs, not directly addressable while your program is active. To reach data outside your own bank pair, you must use enhanced indirect addressing. In addition to the usual two-byte address, the Apple III uses a third byte, the extended byte or X-byte, which determines the bank-pair to be accessed. Instructions like `LDA ($E0),Y` and `STA ($E8),Y` will load and store in the bank-pair determined by the X-bytes corresponding to E0-E1, and E8-E9. The section "Enhanced Indirect Addressing" in *Apple III Pascal Program Preparation Tools* explains this addressing mechanism.

Data declared in Pascal functions and procedures are placed in the Pascal data area, and at startup the X-bytes corresponding to E0-EF are set to point to this bank-pair.

The uncustomized FP6502 and Elems6502 use locations $E0-E1, E2-E3, E4-E5, and E6-E7 to access data passed by address. For each call, data to be passed by address to FP6502 and Elems6502 must be in the same bank-pair. That is, the data addresses must have the same X-byte values. If the only data you pass to FP6502 and Elems6502 are Pascal data, then do nothing with X-bytes, because the system points to Pascal data at startup. If your data are located in other banks, for example in the same bank as your assembly-language program, then you must set the X-bytes ($16E1, 16E3, 16E5, 16E7) to point to your data before making a call to FP6502 or Elems6502. For data local to your program, the X-bytes can be set to 0.

▲ **Warning**
Before returning control to Pascal, reset these X-bytes to their original values.

Be careful how you store the saved X-bytes if your code is reentrant or recursive.

This example shows a good way to pass local data to FP6502.

```
XPGSTART     .EQU       1600              ;X-page start
               .                          ;  X-bytes at $1600
               .
               .
             LDA        XPGSTART+1+0E0    ; X-byte for E0-E1
             PHA                          ;   save to stack
             LDA        XPGSTART+1+0E2    ; X-byte for E2-E3
             PHA                          ;   save to stack
             LDA        XPGSTART+1+0E4    ; X-byte for E4-E5
             PHA                          ;   save to stack
             LDA        XPGSTART+1+0E6    ; X-byte for E6-E7
             PHA                          ;   save to stack
             LDA        #0                ; set X-bytes to 0
             STA        XPGSTART+1+0E0
             STA        XPGSTART+1+0E2
             STA        XPGSTART+1+0E4
             STA        XPGSTART+1+0E6

; FP6502 and Elems6502 calls made after this point now access data
;    in your assembly code.
; To restore the X-bytes to the system values
             PLA                          ; get saved X-byte
             STA        XPGSTART+1+0E6    ;   restore X-byte
             PLA                          ; get saved X-byte
             STA        XPGSTART+1+0E4    ;   restore X-byte
             PLA                          ; get saved X-byte
             STA        XPGSTART+1+0E2    ;   restore X-byte
             PLA                          ; get saved X-byte
             STA        XPGSTART+1+0E0    ;   restore X-byte
```

Assembly-Language Access: SOS

This section describes the use of FP6502 and Elems6502 with SOS.

Files

Use the customizing process described above to produce absolute code versions of FP6502 and Elems6502 with starting addresses and zero-page usage of your choice.

Addressing

FP6502 and Elems6502 each have one entry point. Assembly-language programs call them with the instructions

```
JSR      FP6502
JSR      ELEMS6502
```
(if Elems6502 is required).

Calling programs must include the directives

```
FP6502      EQU      <customized starting address>
ELEMS6502   EQU      <customized starting address>
```

Zero-Page

During customization you specify the zero-page locations used by FP6502 and Elems6502. These locations are used as temporary scratch space and their contents may be arbitrary whenever FP6502 or Elems6502 is called. On the Apple III, there are two classes of zero-page usage. $2C bytes (hex) are needed for scratch work, plus eight more bytes for enhanced indirect addressing. You must manage the X-bytes for these eight locations. FP6502 and Elems6502 use enhanced indirect addressing for all data passed by address. During customization, you will be asked to specify where the two classes of zero-page are to be located. The section "Enhanced Indirect Addressing" in *Apple III Pascal Program Preparation Tools* explains this addressing mechanism.

Loading

Loading the disk images of FP6502 and Elems6502 into memory can be done through calls to SOS. See the *Apple III SOS Reference Manual* for information on SOS calls.

Location of Data

The Apple III CPU can address only 64K bytes at one time, but the machine can handle more memory by either bank switching or enhanced indirect addressing. In order to reach data outside your own bank pair, you will need to use enhanced indirect addressing.

Appendix B

6502 SANE Macros

```
;----------------------------------------------------
;
; FILE:  SANEMACRO.TEXT
;
;  These macros and equates give assembly-language access to
;  the 6502 floating-point arithmetic routines.
;----------------------------------------------------

;----------------------------------------------------
; Operation code masks.
;----------------------------------------------------
FOAOO           .EQU    000     ; add
FOSUB           .EQU    002     ; subtract
FOMUL           .EQU    004     ; multiply
FODIV           .EQU    006     ; divide
FOCMP           .EQU    008     ; compare, no exception from
                                ;    unordered
FOCPX           .EQU    00A     ; compare, signal invalid if
                                ;    unordered
FOREM           .EQU    00C     ; remainder
FOZ2X           .EQU    00E     ; convert to extended
FOX2Z           .EQU    010     ; convert from extended
FOSQRT          .EQU    012     ; square root
FORTI           .EQU    014     ; round to integral value
FOTTI           .EQU    016     ; truncate to integral value
FOSCALB         .EQU    018     ; binary scale
FOLOGB          .EQU    01A     ; binary log
FOCLASS         .EQU    01C     ; classify
FONEXT          .EQU    01E     ; next-after
```

```
FOSETENV        .EQU    001     ; set environment
FOGETENV        .EQU    003     ; get environment
FOSETHV         .EQU    005     ; set halt vector
FOGETHV         .EQU    007     ; get halt vector
FOD2B           .EQU    009     ; convert decimal to binary
FOB2D           .EQU    00B     ; convert binary to decimal
FONEG           .EQU    00D     ; negate
FOABS           .EQU    00F     ; absolute value
FOCPYSGN        .EQU    011     ; copy sign
 ; UNDEFINED      .EQU    013
FOSETXCP        .EQU    015     ; set exception
FOPROCENTRY     .EQU    017     ; procedure-entry
FOPRDCEXIT      .EQU    019     ; procedure-exit
FOTESTXCP       .EQU    01B     ; test exception
 ; UNDEFINED      .EQU    01D
 ; UNDEFINED      .EQU    01F

 ;-----------------------------------------------------------
 ; Operand format masks.
 ;-----------------------------------------------------------
FFEXT           .EQU    00      ; extended  80-bit float
FFDBL           .EQU    01      ; double   64-bit float
FFSGL           .EQU    02      ; single   32-bit float
 ;UNDEFINED      .EQU    03
FFINT           .EQU    04      ; integer   16-bit integer
FFCOMP          .EQU    05      ; comp     64-bit integer
 ; UNDEFINED      .EQU    06
 ; UNDEFINED      .EQU    07

 ;-----------------------------------------------------------
 ; Operation macros: operands should already be on
 ; the stack, with the destination address on top.  Generally
 ; the suffix X, D, S, C, or I determines the format of the
 ; source operand  extended, double, single, comp, or
 ; 16-bit integer.
 ;-----------------------------------------------------------
```

```
;-------------------------------------------------------------
; Auxiliary macros.
;-------------------------------------------------------------
        .MACRO POP      ;16-bit parameter from stack to memory
        PLA
        STA %1
        PLA
        STA %1+1
        .ENDM

        .MACRO PUSH     ;16-bit result from memory to stack
        LDA %1+1
        PHA
        LDA %1
        PHA
        .ENDM

        .MACRO  FOPR    ; push 2-byte opcode, jsr to fp6502
        LDA     #%1
        PHA
        LDA     #%2
        PHA
        JSR     fp6502
        .ENDM

        .MACRO  FOPRO  ; push 0 byte and 1-byte opcode,
                        ; jsr to fp6502
        LDA     #0
        PHA
        LDA     #%1
        PHA
        JSR     fp6502
        .ENDM

        .MACRO  FOPRJ  ; push junk byte and 1-byte opcode,
                        ; jsr to fp6502
        PHA
        LDA     #%1
        PHA
        JSR     fp6502
        .ENDM
```

```
;---------------------------------------------------------------
; Addition.
;---------------------------------------------------------------
        .MACRD  FADDX
        FDPR      FFEXT,FOADD
        .ENDM

        .MACRD  FADDD
        FDPR      FFDBL,FDADD
        .ENDM

        .MACRD  FADDS
        FOPR      FFSGL,FDADD
        .ENDM

        .MACRO  FADDC
        FDPR      FFCOMP,FDADD
        .ENDM

        .MACRO  FADDI
        FDPR      FFINT,FDADD
        .ENDM

;---------------------------------------------------------------
; Subtraction.
;---------------------------------------------------------------
        .MACRD  FSUBX
        FDPR      FFEXT,FDSUB
        .ENDM

        .MACRD  FSUBD
        FDPR      FFDBL,FDSUB
        .ENDM

        .MACRO  FSUBS
        FDPR      FFSGL,FDSUB
        .ENDM

        .MACRD  FSUBC
        FDPR      FFCDMP,FDSUB
        .ENDM

        .MACRD  FSUBI
        FDPR      FFINT,FDSUB
        .ENDM
```

```
;-------------------------------------------------------------
; Multiplication.
;-------------------------------------------------------------
        .MACRO  FMULX
        FOPR     FFEXT,FOMUL
        .ENDM

        .MACRO  FMULD
        FOPR     FFDBL,FOMUL
        .ENDM

        .MACRO  FMULS
        FOPR     FFSGL,FOMUL
        .ENDM

        .MACRO  FMULC
        FOPR     FFCOMP,FDMUL
        .ENDM

        .MACRD  FMULI
        FOPR     FFINT,FDMUL
        .ENDM

;-------------------------------------------------------------
; Division.
;-------------------------------------------------------------
        .MACRD  FDIVX
        FOPR     FFEXT,FODIV
        .ENDM

        .MACRO  FDIVO
        FOPR     FFDBL,FODIV
        .ENDM

        .MACRO  FDIVS
        FOPR     FFSGL,FODIV
        .ENDM

        .MACRO  FDIVC
        FDPR     FFCDMP,FODIV
        .ENDM

        .MACRO  FDIVI
        FOPR     FFINT,FODIV
        .ENDM
```

```
;-----------------------------------------------------------
; Square root.
;-----------------------------------------------------------
        .MACRO   FSQRTX
        FOPRO    FOSQRT
        .ENOM

;-----------------------------------------------------------
; Round to integer, according to the current rounding mode.
;-----------------------------------------------------------
        .MACRO   FRINTX
        FOPRO    FORTI
        .ENOM

;-----------------------------------------------------------
; Truncate to integer, using round toward zero.
;-----------------------------------------------------------
        .MACRO   FTINTX
        FOPRO    FOTTI
        .ENOM

;-----------------------------------------------------------
; Remainder.
;-----------------------------------------------------------
        .MACRO   FREMX
        FOPR      FFEXT,FOREM
        .ENDM

        .MACRO   FREMO
        FOPR      FFOBL,FOREM
        .ENDM

        .MACRO   FREMS
        FOPR      FFSGL,FOREM
        .ENOM

        .MACRO   FREMC
        FOPR      FFCOMP,FOREM
        .ENOM

        .MACRO   FREMI
        FOPR      FFINT,FOREM
        .ENOM
```

```
;-------------------------------------------------------
; Logb.
;-------------------------------------------------------
        .MACRD   FLDGBX
        FDPRO    FDLDGB
        .ENDM

;-------------------------------------------------------
; Scalb.
;-------------------------------------------------------
        .MACRD   FSCALBX
        FOPR       FFINT,FDSCALB
        .ENDM

;-------------------------------------------------------
; Copy-sign.
;-------------------------------------------------------
        .MACRD   FCPYSGNX
        FDPR       FFEXT,FOCPYSGN
        .ENDM

        .MACRO   FCPYSGND
        FDPR       FFDBL,FDCPYSGN
        .ENDM

        .MACRO   FCPYSGNS
        FOPR       FFSGL,FOCPYSGN
        .ENDM

        .MACRD   FCPYSGNC
        FDPR       FFCDMP,FDCPYSGN
        .ENDM

        .MACRD   FCPYSGNI
        FDPR       FFINT,FDCPYSGN
        .ENDM

;-------------------------------------------------------
; Negate.
;-------------------------------------------------------
        .MACRD   FNEGX
        FOPRO    FDNEG
        .ENDM
```

```
;------------------------------------------------------------
; Absolute value.
;------------------------------------------------------------
       .MACRO   FABSX
       FOPRO    FOABS
       .ENDM

;------------------------------------------------------------
; Next-after.  NOTE:  both operands are of the
; the same format, as specified by the usual suffix.
;------------------------------------------------------------
       .MACRO   FNEXTS
       FOPR      FFSGL,FONEXT
       .ENOM

       .MACRO   FNEXTO
       FOPR      FFOBL,FONEXT
       .ENOM

       .MACRO   FNEXTX
       FOPR      FFEXT,FONEXT
       .ENOM

;------------------------------------------------------------
; Conversion to extended.
;------------------------------------------------------------
       .MACRO   FX2X
       FOPR      FFEXT,FOZ2X
       .ENOM

       .MACRO   FO2X
       FOPR      FFOBL,FOZ2X
       .ENOM

       .MACRO   FS2X
       FOPR      FFSGL,FOZ2X
       .ENOM

       .MACRO   FI2X

           ; 16-bit integer, by address
       FOPR      FFINT,FOZ2X
       .ENOM

       .MACRO   FC2X
       FOPR      FFCOMP,FOZ2X
       .ENOM
```

```
;---------------------------------------------------------
; Conversion from extended.
;---------------------------------------------------------
       .MACRD  FX2D
       FDPR      FFDBL,FDX2Z
       .ENDM

       .MACRD  FX2S


       FDPR      FFSGL,FOX2Z
       .ENDM

       .MACRD  FX2I
       FDPR      FFINT,FDX2Z
       .ENDM

       .MACRO  FX2C
       FOPR      FFCDMP,FDX2Z
       .ENDM

;-------------------------------------------------------
; Binary to decimal conversion.
;---------------------------------------------------------
       .MACRO  FX2DEC
       FDPR      FFEXT,FDB2D
       .ENOM

       .MACRD  FD2DEC
       FDPR      FFDBL,FDB2O
       .ENDM

       .MACRD  FS2DEC
       FDPR      FFSGL,FOB2D
       .ENDM

       .MACRD  FC2DEC
       FDPR      FFCDMP,FDB2D
       .ENDM

       .MACRD  FI2OEC
       FDPR      FFINT,FDB2D
       .ENDM
```

Appendix C

6502 SANE Quick Reference Guide

This guide contains diagrams of the SANE data formats and the 6502 SANE operations and environment word.

Formats of SANE Types

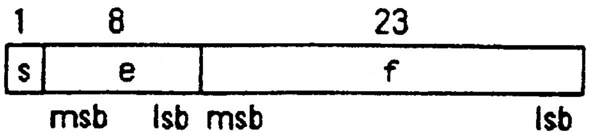
Each of the diagrams below is followed by the rules for evaluating the number v.

In each field of each diagram, the leftmost bit is the msb and the rightmost is the lsb. The 6502 engine uses the convention that least significant bytes are stored in low memory.

Table C-1. *Format Diagram Symbols*

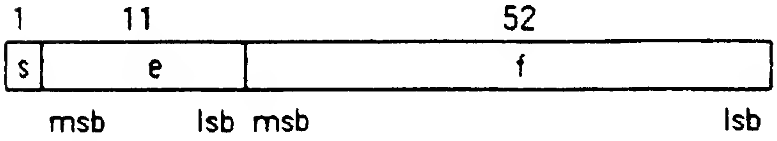
v	value of number
s	sign bit
e	biased exponent
i	explicit one's-bit (extended type only)
f	fraction

Single: 32 Bits



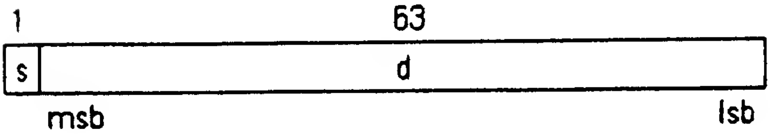
If $0 < e < 255$,	then $v = (-1)^s * 2^{(e-127)} * (1.f)$.
If $e = 0$ and $f \neq 0$,	then $v = (-1)^s * 2^{(-126)} * (0.f)$.
If $e = 0$ and $f = 0$,	then $v = (-1)^s * 0$.
If $e = 255$ and $f = 0$,	then $v = (-1)^s * \infty$.
If $e = 255$ and $f \neq 0$,	then v is a NaN.

Double: 64 Bits



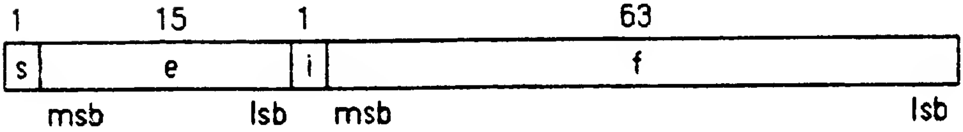
If $0 < e < 2047$,	then $v = (-1)^s * 2^{(e-1023)} * (1.f)$.
If $e = 0$ and $f \neq 0$,	then $v = (-1)^s * 2^{(-1022)} * (0.f)$.
If $e = 0$ and $f = 0$,	then $v = (-1)^s * 0$.
If $e = 2047$ and $f = 0$,	then $v = (-1)^s * \infty$.
If $e = 2047$ and $f \neq 0$,	then v is a NaN.

Comp: 64 Bits



If s = 1 and d = 0, then v is the unique comp NaN.
Otherwise, v is the two's-complement value of the 64-bit representation.

Extended: 80 Bits



If 0 < = e < 32767, then v = $(-1)^s * 2^{(e-16383)} * (i.f)$.
If e = 32767 and f = 0, then v = $(-1)^s * \infty$, regardless of
If e = 32767 and f ≠ 0, then v is a NaN, regardless of i.

Operations

In the operations below, the operation's mnemonic is followed by the opword in parentheses: the first byte is the operand format code; the second is the operation code. For some operations, the first byte (xx) of the opword is ignored.

Abbreviations and Symbols

The symbols and abbreviations in this section closely parallel those in the text, although some are shortened. In some cases, the same symbol has various meanings, depending on context.

Operands

DST	destination operand (passed by address)
SRC	source operand passed by address
SRC	source operand passed by value
SRC2	second source operand passed by address

Push SRC before DST. Push SRC2 before SRC.

Data Types

X	extended (80 bits, passed by address)
D	double (64 bits, passed by address)
S	single (32 bits, passed by address)
I	integer (16 bits) passed by address
I	integer (16 bits) passed by value
C	comp (64 bits, passed by address)
Dec	decimal record
Decform	decform record

6502 Processor Registers

Xreg	X register
Yreg	Y register
Preg	processor status register
Nbit	negative bit of processor status register
Zbit	zero bit of processor status register

Exceptions

Xcps	exceptions, collectively
I	invalid operation
U	underflow
O	overflow
D	divide-by-zero
X	inexact

For each operation, an exception marked with x indicates that the operation will signal the exception for some input.

Environment and Halts

EnvWrd	SANE environment word (16-bit integer)
HltVctr	SANE halt vector (16-bit integer)

Arithmetic Operations and Auxiliary Routines (Entry Point FP6502)

Operation	Operands and Data Types				Exceptions				
ADD	DST	←	DST	+ SRC	I	U	O	D	X
FADDX (0000)	X		X	X	x	-	x	-	x
FADDD (0100)	X		X	D	x	-	x	-	x
FADDS (0200)	X		X	S	x	-	x	-	x
FADDC (0500)	X		X	C	x	-	x	-	x
FADDI (0400)	X		X	I	x	-	x	-	x
SUBTRACT	DST	←	DST	- SRC	I	U	O	D	X
FSUBX (0002)	X		X	X	x	-	x	-	x
FSUBD (0102)	X		X	D	x	-	x	-	x
FSUBS (0202)	X		X	S	x	-	x	-	x
FSUBC (0502)	X		X	C	x	-	x	-	x
FSUBI (0402)	X		X	I	x	-	x	-	x
MULTIPLY	DST	←	DST	* SRC	I	U	O	D	X
FMULX (0004)	X		X	X	x	x	x	-	x
FMULD (0104)	X		X	D	x	x	x	-	x
FMULS (0204)	X		X	S	x	x	x	-	x
FMULC (0504)	X		X	C	x	-	x	-	x
FMULI (0404)	X		X	I	x	-	x	-	x
DIVIDE	DST	←	DST	/ SRC	I	U	O	D	X
FDIVX (0006)	X		X	X	x	x	x	x	x
FDIVD (0106)	X		X	D	x	x	x	x	x
FDIVS (0206)	X		X	S	x	x	x	x	x
FDIVC (0506)	X		X	C	x	x	-	x	x
FDIVI (0406)	X		X	I	x	x	-	x	x

Operation	Operands and Data Types					Exceptions				
SQUARE ROOT	DST	←	sqrt(DST)			I	U	O	D	X
FSQRTX (0012)	X		X			x	-	-	-	x
ROUND TO INT	DST	←	rnd(DST)			I	U	O	D	X
FRINTX (0014)	X		X			x	-	-	-	x
TRUNC TO INT	DST	←	chop(DST)			I	U	O	D	X
FTINTX (0016)	X		X			x	-	-	-	x
REMAINDER	DST	←	DST	REM	SRC	I	U	O	D	X
FREMX (000C)	X		X		X	x	-	-	-	-
FREMD (010C)	X		X		D	x	-	-	-	-
FREMS (020C)	X		X		S	x	-	-	-	-
FREMC (050C)	X		X		C	x	-	-	-	-
FREMI (040C)	X		X		I	x	-	-	-	-

Xreg	←	7 low-order bits of |n|
Nbit	←	sign bit of n
Yreg	←	00 if sign bit of n is 0
	←	80 if sign bit of n is 1

(n is the integer quotient DST / SRC.)

Operation	Operands and Data Types					Exceptions				
LOG BINARY	DST	←	logb(DST)			I	U	O	D	X
FLOGBX (001A)	X		X			x	-	-	x	-
SCALE BINARY	DST	←	DST * 2 ^ **SRC**			I	U	O	D	X
FSCALBX (0018)	X		X	I		x	x	x	-	x
NEGATE	DST	←	-DST			I	U	O	D	X
FNEGX (000D)	X		X			-	-	-	-	-
ABSOLUTE VALUE	DST	←	|DST|			I	U	O	D	X
FABSX (000F)	X		X			-	-	-	-	-
COPY-SIGN	DST	←	DST with	SRC's sign		I	U	O	D	X
FCPYSGNX (0011)	X		X	X		-	-	-	-	-
FCPYSGND (0111)	X		X	D		-	-	-	-	-
FCPYSGNS (0211)	X		X	S		-	-	-	-	-
FCPYSGNC (0511)	X		X	C		-	-	-	-	-
FCPYSGNI (0411)	X		X	I		-	-	-	-	-
NEXT-AFTER	DST	←	next after DST	toward SRC		I	U	O	D	X
FNEXTX (001E)	X		X	X		x	x	x	-	x
FNEXTD (011E)	D		D	D		x	x	x	-	x
FNEXTS (021E)	S		S	S		x	x	x	-	x

Conversions (Entry Point FP6502)

Operation	Operands and Data Types				Exceptions				
CONVERT									
Bin to Bin	DST	←	SRC		I	U	O	D	X
FX2X (0010)	X		X		x	-	-	-	-
FX2D (0110)	D		X		x	x	x	-	x
FX2S (0210)	S		X		x	x	x	-	x
FX2C (0510)	C		X		x	-	-	-	x
FX2I (0410)	I		X		x	-	-	-	x
FX2D (010E)	X		D		x	-	-	-	-
FX2S (020E)	X		S		x	-	-	-	-
FX2C (050E)	X		C		-	-	-	-	-
FX2I (040E)	X		I		-	-	-	-	-
Bin to Dec	DST	←	SRC according to	SRC2	I	U	O	D	X
FX2DEC (000B)	Dec		X	Decform	x	-	-	-	x
FD2DEC (010B)	Dec		D	Decform	x	-	-	-	x
FS2DEC (020B)	Dec		S	Decform	x	-	-	-	x
FC2DEC (050B)	Dec		C	Decform	-	-	-	-	x
FI2DEC (040B)	Dec		I	Decform	-	-	-	-	x

(First SRC2 is pushed, then SRC, then DST.)

Dec to Bin	DST	←	SRC	I	U	O	D	X
FDEC2X (0009)	X		Dec	-	x	x	-	x
FDEC2D (0109)	D		Dec	-	x	x	-	x
FDEC2S (0209)	S		Dec	-	x	x	-	x
FDEC2C (0509)	C		Dec	x	-	-	-	x
FDEC2I (0409)	I		Dec	x	-	-	-	x

Compare and Classify (Entry Point FP6502)

Operation	Operands and Data Types		Exceptions				
COMPARE							
No invalid *for unordered*	Preg, Xreg, Yreg ← <relation> where SRC <relation>	DST	I	U	O	D	X
FCMPX (0008)	X	X	x	-	-	-	-
FCMPD (0108)	D	X	x	-	-	-	-
FCMPS (0208)	S	X	x	-	-	-	-
FCMPC (0508)	C	X	x	-	-	-	-
FCMPI (0408)	I	X	x	-	-	-	-

(Invalid only for signaling NaN inputs.)

Signal invalid *if unordered*	Preg, Xreg, Yreg ← <relation> where SRC <relation>	DST	I	U	O	D	X
FCPXX (000A)	X	X	x	-	-	-	-
FCPXD (010A)	D	X	x	-	-	-	-
FCPXS (020A)	S	X	x	-	-	-	-
FCPXC (050A)	C	X	x	-	-	-	-
FCPXI (040A)	I	X	x	-	-	-	-

<relation>	**P Register**			**X Register**	**Y Register**
	Z	**N**	**V**		
SRC > DST	0	0	1	40	40
SRC < DST	0	1	0	80	80
SRC = DST	1	0	0	02	00
SRC & DST unordered	0	0	0	01	01

CLASSIFY	Xreg, Yreg, Nbit ←	SRC	I	U	O	D	X
FCLASSX (001C)		X	-	-	-	-	-
FCLASSD (011C)		D	-	-	-	-	-
FCLASSS (021C)		S	-	-	-	-	-
FCLASSI (041C)		I	-	-	-	-	-
FCLASSC (051C)		C	-	-	-	-	-

Class	**Xreg**	**Sign**	**Yreg**	**Nbit**
signaling NaN	FC	positive	00	0
quiet NaN	FD	negative	80	1
infinite	FE			
zero	FF			
normalized	00			
denormalized	01			

Environmental Control (Entry Point FP6502)

Operation	**Operands and Data Types**	**Exceptions**

GET ENVIRONMENT
FGETENV (xx03)
	Xreg ← low byte of EnvWrd
	Yreg ← high byte of EnvWrd
		I U O D X
		- - - - -

SET ENVIRONMENT FSETENV (xx01)	EnvWrd ← **SRC** I	I U O D X - - - - -

(Exceptions set by set-environment cannot cause halts.)

TEST EXCEPTION FTESTXCP (xx1B)	Zbit ← **SRC** Xcps clear I	I U O D X - - - - -
SET EXCEPTION FSETXCP (xx15)	EnvWrd ← EnvWrd AND **SRC** I	I U O D X x x x x x
PROCEDURE ENTRY FPROCENTRY (xx17)	DST ← EnvWrd, EnvWrd ← 0 I	I U O D X - - - - -
PROCEDURE EXIT FPROCEXIT (xx19)	EnvWrd ← **SRC** AND current Xcps I	I U O D X x x x x x

Halt Control (Entry Point FP6502)

SET HALT VECTOR FSETHV (xx05)	HltVctr ← **SRC** I	I U O D X - - - - -

GET HALT VECTOR
FGETHV (xx07)
	Xreg ← low byte of HltVctr
	Yreg ← high byte of HltVctr
		I U O D X
		- - - - -

Elementary Functions (Entry Point ELEMS6502)

Operation	Operands and Data Types	Exceptions
BASE-E LOGARITHM FLNX (xx00)	DST ← ln(DST) X X	I U O D X x - - x x
BASE-2 LOGARITHM FLOG2X (xx02)	DST ← log2(DST) X X	I U O D X x - - x x
BASE-E LOG1 (LN1) FLN1X (xx04)	DST ← ln(1 + DST) X X	I U O D X x x - x x
BASE-2 LOG1 FLOG21X (xx06)	DST ← log2(1 + DST) X X	I U O D X x x - x x
BASE-E EXPONENTIAL FEXPX (xx08)	DST ← e^DST X X	I U O D X x x x - x
BASE-2 EXPONENTIAL FEXP2X (xx0A)	DST ← 2^DST X X	I U O D X x x x - x
BASE-E EXP1 FEXP1X (xx0C)	DST ← e^DST - 1 X X	I U O D X x x x - x
BASE-2 EXP1 FEXP21X (xx0E)	DST ← 2^DST - 1 X X	I U O D X x x x - x
INTEGER EXPONENTIATION FXPWRI (xx10)	DST ← DST^SRC X X I	I U O D X x x x x x
GENERAL EXPONENTIATION FXPWRY (xx12)	DST ← DST^SRC X X X	I U O D X x x x x x
COMPOUND INTEREST FCOMPOUND (xx14)	DST ← compound(SRC2,SRC) X X X	I U O D X x x x x x

(SRC2 is the rate; SRC is the number of periods.)

Operation	Operands and Data Types	Exceptions
ANNUITY FACTOR FANNUITY (xx16)	DST ← annuity(SRC2,SRC) X X X	I U O D X x x x x x

(SRC2 is the rate; SRC is the number of periods.)

Operation	Operands and Data Types	Exceptions
ARCTANGENT ATANX (xx18)	DST ← atan(DST) X X	I U O D X x x - - x
SINE FSINX (xx1A)	DST ← sin(DST) X X	I U O D X x x - - x

COSINE FCOSX (xx1C)	DST X	←	cos(DST) X	I x	U x	O -	D -	X x
TANGENT FTANX (xx1E)	DST X	←	tan(DST) X	I x	U x	O -	D x	X x
RANDOM FRANDX (xx20)	DST X	←	random(DST) X	I x	U x	O x	D -	X x

(No exceptions, for valid arguments.)

Environment Word

The floating-point environment is encoded in the 16-bit integer format as shown below in hexadecimal.

msb lsb

r	r	–	x	d	o	u	i	R	R	–	X	D	O	U	I

rounding direction | exception flags | rounding precision | halts enabled

rounding direction, bits C000			rr
0000	—	to-nearest	
4000	—	upward	
8000	—	downward	
C000	—	toward-zero	
exception flags, bits 1F00			
0100	—	invalid	i
0200	—	underflow	u
0400	—	overflow	o
0800	—	division-by-zero	d
1000	—	inexact	x

rounding precision, bits 00C0			RR
0000	—	extended	
0040	—	double	
0080	—	single	
00C0	—	UNDEFINED	
halts enabled, bits 001F			
0001	—	invalid	I
0002	—	underflow	U
0004	—	overflow	O
0008	—	division-by-zero	D
0010	—	inexact	X

Bits 2000 and 0020 are not used.

Note that the default environment is represented by the integer value zero.

Index